IDA PAPER P-2628

AN EXAMINATION OF SELECTED COMMERCIAL SOFTWARE TESTING TOOLS

Bill R. Brykczynski, *Task Leader*
Christine Youngblut
Reginald N. Meeson

October 1991

Prepared for
Strategic Defense Initiative Organization

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311-1772

IDA Log No. HQ 91-039501

DEFINITIONS

IDA publishes the following documents to report the results of its work.

Reports

Reports are the most authoritative and most carefully considered products IDA publishes. They normally embody results of major projects which (a) have a direct bearing on decisions affecting major programs, (b) address issues of significant concern to the Executive Branch, the Congress and/or the public, or (c) address issues that have significant economic implications. IDA Reports are reviewed by outside panels of experts to ensure their high quality and relevance to the problems studied, and they are released by the President of IDA.

Group Reports

Group Reports record the findings and results of IDA established working groups and panels composed of senior individuals addressing major issues which otherwise would be the subject of an IDA Report. IDA Group Reports are reviewed by the senior individuals responsible for the project and others as selected by IDA to ensure their high quality and relevance to the problems studied, and are released by the President of IDA.

Papers

Papers, also authoritative and carefully considered products of IDA, address studies that are narrower in scope than those covered in Reports. IDA Papers are reviewed to ensure that they meet the high standards expected of refereed papers in professional journals or formal Agency reports.

Documents

IDA Documents are used for the convenience of the sponsors or the analysts (a) to record substantive work done in quick reaction studies, (b) to record the proceedings of conferences and meetings, (c) to make available preliminary and tentative results of analyses, (d) to record data developed in the course of an investigation, or (e) to forward information that is essentially unanalyzed and unevaluated. The review of IDA Documents is suited to their content and intended use.

The work reported in this document was conducted under contract MDA 903 89 C 0003 for the Department of Defense. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that Agency.

This Paper has been reviewed by IDA to assure that it meets high standards of thoroughness, objectivity, and appropriate analytical methodology and that the results, conclusions and recommendations are properly supported by the material presented.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 1991	Final

4. TITLE AND SUBTITLE
An Examination of Selected Commercial Software Testing Tools

5. FUNDING NUMBERS
MDA 903 89 C 0003

Task T-R2-597.21

6. AUTHOR(S)
Bill R. Brykczynski, Christine Youngblut, Reginald N. Meeson

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Defense Analyses (IDA)
1801 N. Beauregard St.
Alexandria, VA 22311-1772

8. PERFORMING ORGANIZATION REPORT NUMBER
IDA Paper P-2628

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Computer Resources, Engineering Division
SDIO
Room 1E149, The Pentagon
Washington, D.C. 20301-7100

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, unlimited distribution: 29 October 1991.

12b. DISTRIBUTION CODE
2A

13. ABSTRACT (Maximum 200 words)

This paper reports on the examination of ten commercial tools identified for the static and dynamic analysis of Ada code. It provides software development managers with information that may help them gain an understanding of the types of tools that are commercially available, the functionality of these tools, and how they can aid the development of Ada software. During the course of the assessment, the tools were applied to a series of Ada programs in order to assess their functionality. Each tool was then described in terms of its functionality, ease of use, and documentation and support. Problems encountered during the examination and other pertinent observations were also recorded. Significant findings from this study include the following: 1) some of the examined tools could be brought into immediate use to improve the cost-effectiveness of testing for SDI software development efforts, 2) the coverage analyzers provide reporting data that can support the management of SDI testing efforts, and 3) many of the tools can be used in conjunction to overcome the limitations of particular tools.

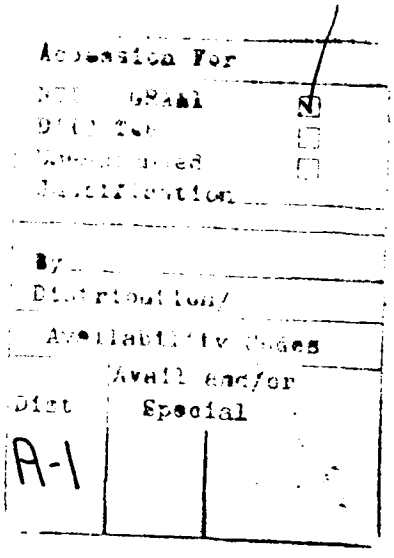
14. SUBJECT TERMS	15. NUMBER OF PAGES
Software Testing; Software Testing Tools; Ada; SDI.	108
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

IDA PAPER P-2628

AN EXAMINATION OF SELECTED COMMERCIAL SOFTWARE TESTING TOOLS

Bill R. Brykczynski, *Task Leader*
Christine Youngblut
Reginald N. Meeson



October 1991

INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 89 C 0003
Task T-R2-597.21

PREFACE

This paper was prepared by the Institute for Defense Analyses (IDA) for the Strategic Defense Initiative Organization (SDIO), under contract MDA 903 89 C 0003, Subtask Order T-R2-597.21, "SDS Test and Evaluation." The objective of the subtask is to assist the SDIO in planning, executing, and monitoring software testing and evaluation research, development, and practice.

In support of this objective, IDA conducted an examination of several commercially available tools that support static and dynamic analysis of Ada code. This paper presents the results of the assessment and provides software development managers with information on current capabilities of commercial testing tools.

This paper was reviewed by the following members of the IDA research staff: Dr. Robert Atwell, Dr. Dennis Fife, Dr. Randy Garrett, Dr. Karen Gordon, Ms. Audrey Hook, and Dr. Richard J. Ivanetich.

EXECUTIVE SUMMARY

Software testing is labor intensive and can consume over 50% of software development costs. Rarely is sufficient, effective testing performed as evidenced by the fact that a failure rate of 3 to 10 failures per thousand lines of code is typical for commercial software. Moreover, the cost of correcting a defect increases as software development progresses; for example, the cost of fixing a requirements fault during operation can be 60 to 100 times the cost of fixing that same fault during early development stages. Consequently, timely defect detection is important. Automated testing tools can alleviate these problems by reducing the traditionally manual nature of testing and encouraging the application of improved testing practices.

Over one hundred testing tools from nearly seventy vendors of testing tools were identified. A short list of tools supporting the static and dynamic analysis of Ada code was developed. Consideration of tools that are limited to quality analysis or regression analysis, dependendent on special hardware, or form part of a computer-aided software engineering system was postponed and these tools excluded from the list. From the short list, ten tools that support the testing of Ada code were selected for examination: MALPAS, SoftTest, S-TCAT/Ada, TCAT/Ada, TCAT-PATH, TDGen, TSCOPE, TBGEN, TCMON, and TestGen. During the course of the examination, the tools were applied to a series of Ada programs in order to assess their functionality. Each tool was then described in terms of its functionality, ease of use, and documentation and support. Problems encountered during tool use and other pertinent observations were also recorded.

Compared to other software development tools for Ada, the market offers relatively few commercial tools that support the static and dynamic analysis needed for testing Ada code. Significant findings from this study include the following:

- Some of the examined tools could be brought into immediate use to improve the cost-effectiveness of testing for SDI software development efforts.
- The coverage analyzers provide reporting data that can support the management of SDI testing efforts.

This report provides software development managers with information that may help them gain an understanding of the types of software testing tools that are commercially available and how these tools can aid the development of Ada software.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1 Purpose and Scope

This report provides software development managers with information that may help them gain an understanding of the types of software testing tools that are commercially available, the functionality of these tools, and how they can aid the development of Ada software.

Tools are available to support a variety of testing tasks at different stages in the software life cycle. To make best use of available resources, the work described here was limited to the examination of tools that support the static and dynamic analysis needed for testing Ada code. Code-based testing was selected as being one area where automated support is critically needed both to increase software reliability and to reduce development costs. Restriction to the Ada programming language [ANSI/MIL-STD-1983] was adopted in view of Department of Defense (DoD) Directive 5000.1, which requires the use of Ada for weapons systems [DoDD 1991]. The Strategic Defense System Initiative Organization (SDIO) [SDI 1991] also requires use of Ada. It is expected, therefore, that the results of this work will apply to the majority of Strategic Defense Initiative (SDI) software development efforts.

Section 2 of this report describes how particular tools were selected for examination and the method of examination. Details of the capabilities of these tools and observations made during the examinations are given in Section 3. The findings resulting from this work are reported in Section 4. Appendix A lists all the commercial testing tools that were identified. Sample outputs from each examined tool are provided in Appendices B through F.

1.2 Background

Definitions of some relevant terms and testing statistics will help to clarify the scope of this work and the applicability of the findings to SDI software development efforts.

An *error* is a mistake made by a software developer. Its manifestation may be a textual problem in the code called a *fault* or *defect*. A *failure* occurs when an encountered fault prevents software from performing a required function within specified limits.

Testing refers to the act of detecting the presence of faults, or demonstrating their absence, and is distinguished from *debugging* where faults are textually isolated. Three stages of testing are commonly recognized; these are *unit* testing, *integration testing*, and *system testing*. In the first of these, unit testing, each program module is tested in isolation. In integration testing, these modules are combined so that successively larger groups of integrated software and hardware modules can be tested. Finally, system testing examines an integrated hardware and software system to verify that the system meets its specified requirements.

In *bottom-up testing* the modules at the bottom of the invocation hierarchy are tested independently, then modules at the next higher level that call these modules, and so on. *Top-down testing* starts at the highest-level module, with stubs replacing the modules it invokes. These stubs are then replaced by the next lower-level modules, with new stubs being provided for modules these call, and so on.

This report looks at unit and integration testing by both dynamic and static analysis. *Dynamic analysis* approaches rely on executing a piece of software with selected test data. The effectiveness of any dynamic analysis technique is directly related to the test data used. Current tools attempt to detect faults rather than demonstrate the absence of faults. Additionally, these tools can only detect faults whose effects propagate to software outputs. *Static analysis* approaches do not require software execution and can demonstrate the absence of certain types of defects such as variable typing errors. The absence of execution, however, means that they cannot detect faults that depend on the underlying operating environment. Consequently, effective testing requires a combination of static and dynamic analysis approaches. The key types of dynamic and static analysis tools are identified in Tables 1 and 2, respectively. Table 2 omits tools for analyses routinely performed by Ada compilers, such as type analysis.

For dynamic analysis, different strategies or heuristics can be used to drive test data generation. Commercial automated support is currently only available for *functional* and *structural* strategies. In the first case, test data is derived from the program's requirements with no regard to program structure; these approaches are language-independent. In structural strategies, test data is derived solely from the program structure where this structure is often represented as a directed graph (digraph). The only functional strategy currently supported is *cause-effect graphing*. Here causes relate to distinct input conditions or equivalence classes of input conditions, and effects relate to the resulting output conditions or system transformations. Test data is derived from a combinational logic network that represents the logical relationships between causes and effects. The structural strategies at the unit level that are supported are *branch* and *path* testing. Branch testing requires each conditional branch statement and the code segment whose

TABLE 1. Tools to Support Dynamic Testing

Type	Description
Assertion Analyzer	During execution, evaluates logical expressions inserted into the software that specify required program states or conditions on variables.
Code Instrumentor	Inserts probes, such as instructions or assertions, into a program to aid statement or resource monitoring, or other activities.
Coverage Analyzer	Assesses measures associated with the execution of program structural elements to determine the adequacy of a series of test runs.
Keystroke Capture	Captures keystroke sequences for automatic playback of test sessions.
Mutation Analyzer	Determines test set thoroughness by measuring the extent to which a test set can discriminate between a program and fault-simulating variants.
Performance Analyzer	Measures the ability of a (sub)system to perform its functions within speed and storage allocation constraints.
Oracle	Produces correct outputs to compare with actual software outputs.
Regression Tester	Retests software to verify that modifications have not caused unintended effects and that the software still meets specified requirements.
Reliability Analyzer	Determines achieved level of reliability of an existing system (component) based on the rate of defect detection.
Test Data Generator	Uses a test strategy to generate test data in a methodical manner. The ideal is to find the minimal set of test cases that result in discovery of a maximal set of defects.
Test Manager	Controls a large and evolving amount of information on system features to be tested, as well as test plans, test data, and test results.
Test Bed	Directs the execution of the software under test against a collection of test data sets. Usually it records and organizes the output generated.
Trace Analyzer	Provides a record of program execution; it states the sequence in which instructions were executed.

TABLE 2. Tools to Support Static Analysis

Type	Description
Code Auditor	Checks for conformance to prescribed programming standards and practices.
Concurrency Analyzer	Determines synchronization patterns in a concurrent program.
Cross-Reference Analyzer	Provides cross-reference information on system components, e.g., data name, statement label, literal use, inter-subroutine call cross-indexes.
Data Flow Analyzer	Performs graphical analysis of collections of sequential data definitions and reference patterns to determine constraints that can be placed on data values at various execution points.
Expression Analyzer	Detects certain commonly occurring faults associated with evaluation of expressions.
Interface Analyzer	Checks the interfaces between program units for consistency and adherence to predefined rules or axioms.
Metric Analyzer	Measures the extent or degree to which a product possesses and exhibits a certain quality, property, or attribute.
Path Analyzer	Identifies all possible paths through a program to detect incomplete paths, unexecutable paths, or conditions that drive path execution.
Reference Analyzer	Detects reference anomalies, e.g., when a variable is referenced along a program path before it is assigned a value.
Safety Analyzer	Identifies the possible causes, and consequence, of critical system failures and so determines the necessary fault tolerance or other mechanisms needed to ensure safe operation under various operating conditions.
Symbolic Evaluator	Accepts symbolic values for some program inputs and algebraically manipulates them according to the expressions in which they appear.
Unit Analyzer	Determines whether or not the units or physical dimensions attributed to an entity are correctly defined and consistently used.
Update Analyzer	Compares two versions of a module to look for differences.

execution is controlled by this conditional to be executed at least once. Similarly, path testing requires execution of every path conditional and associated code segment. Path testing is the more stringent strategy but can incur unacceptable computational costs. The equivalent structural strategy at the integration level requires that each pair of module invocations is executed at least once.

Testing is labor-intensive and can consume over 50% of software development costs. In one particular case, NASA's Apollo program, 80% of the total software development effort was incurred by testing [Dunn 1984]. In general, schedule pressure limits the amount of testing that is performed and defects frequently lead to the failure of operational software. For example, 3 to 10 failures per thousand lines of code (KLOC) are typical for commercial software and 1 to 3 failures for industrial software [Boehm 1988]. With a rate of 0.1 failures per KLOC after delivery for its shuttle code, however, NASA has demonstrated that poor reliability can be avoided [Myers 1988]. The cost of correcting a defect increases as software development progresses; for example, the cost of fixing a requirements fault during operation can be 60 to 100 times the cost of fixing that same fault during early development stages [Pressman 1987]. Consequently, timely defect detection is important.

Automated tools can improve testing cost effectiveness; indeed, they are a prerequisite for most static analysis approaches. In addition to eliminating some repetitive manual tasks, tools can promote effective dynamic analysis by guiding the selection of test data and monitoring test executions. Through capturing and reporting data gathered during the performance of testing activities, tools also support the quantitative process measurement that is necessary for controlling the testing process. Benefits claimed by some of the tools discussed later include:[1]

- Tool X can save a developer $15,000 or more per KLOC.
- Tool Y has given clients an 8:1 reduction in test development effort and one client has achieved a reduction from 1.3 to 0.072 failures per 1,000 lines of source code.

Of course, testing tools are not the only mechanism for improving software reliability. Software inspections, for example, have been reported to find 60% to 90% of software defects, while reducing total development costs by as much as 25% [Fagan 1986]. These and other approaches are discussed in [Brykczynski 1990].

1. Tool names are omitted since these claims have been neither validated, nor invalidated, in the course of this work.

2. APPROACH AND METHODS

The overall approach taken to this work was to identify vendors of testing tools, select tools for examination, and apply the selected tools in the testing of sample pieces of code. These activities are described below.

2.1 Vendor Identification

Tool vendors were identified from a number of sources, specifically:

- The tools fair reported in IEEE *Software* [Lutz 1990].
- The survey of Ada tool and service suppliers in *Defense Science* [DefSci 1990].
- IDA's survey of the state of the art in software testing and analysis [Youngblut 1989].
- Input from the SDI Cooperative Research Exchange (SCORE) effort.
- Tools exhibited at the Tri-Ada Conference, held October 1990 in Baltimore.
- The tools fair at the 8th International Conference on Testing Computer Software, held July 1991 in Washington D.C.

Nearly seventy vendors of over a hundred tools were identified. A list of these is given in Table A-1 in Appendix A. A short list of those tools supporting static and dynamic analysis of Ada code was prepared and tool information sought from the vendors. In several cases, vendors gave in-house demonstrations of their tools.

2.2 Tool Selection

Additional criteria were applied to refine the list to be compatible with the resources available for tool examination. To ensure that the results apply to the largest possible audience, it was determined that selected tools should be essentially independent of processor architecture. Consequently, non-intrusive monitors for real-time systems, which require special purpose hardware, were not considered. For a similar reason, tools tied to a particular Ada compiler were not considered. Because the relationship between quality metrics and software reliability is not well understood, tools restricted to quality analysis were also not considered. Examination of testing tools that are part of a computer-aided software engineering (CASE) system or that perform regression testing was

postponed for a later effort.

The following tools were selected:

- Malvern Program Analysis System (MALPAS). MALPAS provides static analysis capabilities suitable for unit testing and bottom-up integration testing. It utilizes an intermediate language (IL). Translators exist for several languages including Pascal, Fortran, and C.
- SoftTest. This is a requirements-based testing tool that uses cause-effect graphing to generate test conditions. It is independent of programming languages.
- TCAT/Ada, TCAT-PATH, S-TCAT/Ada, TDGen, and TSCOPE. These tools provide structural coverage analysis at the unit and integration levels, test data generation, and animation of the increasing levels of coverage achieved. Versions of the tools supporting Ada, C, Cobol, Fortran, and Pascal programming languages are available.
- Test Bed Generator (TBGEN) and Test Coverage Monitor/Program Bottleneck Finder (TCMON). TBGEN supports dynamic testing of Ada code at the unit and integration levels by generating a testbed that allows a user to control subprogram execution and observe the results. TCMON allows a user to study the coverage of test data or analyze the dynamic behavior of an Ada program. These two tools can be used independently or combined to generate a monitored testbed.
- TestGen. Part of the Ada Integrated Software Lifecycle Environment (AISLE) family of Ada design tools, TestGen provides three types of structural module coverage analysis for testing Ada code at the unit level.

Two additional selected tools whose examinations were postponed are:

- AdaQuest. A static and dynamic testing system based on an existing verification and validation system for Fortran. The first complete version of AdaQuest was due for release in May 1991 but is unavailable as of this writing.
- T. A new version of this functional test data generation tool that includes extensive additions is due to be released in September 1991. A testbed that drives program execution is expected to be released at the same time. Examination of T has been postponed until these become available.

2.3 Method of Examination

Each tool was used to test several small Ada programs. The goal of these initial tool applications was to allow the examiner to gain familiarity with overall tool operation. Each tool was subsequently applied to the same Ada program. This software was the Ada Lexical Analyzer Generator program that creates a lexical analyzer or "next-token" procedure for use in a compiler, pretty printer, or other language processing program [Meeson 1989]. It was developed for the Software Technology for Adaptable, Reliable Systems (STARS) program and consists of several Ada subprograms with a total of 3,253 lines of code.

The experience gained by installing and using the tools was used to prepare a short review of each tool. Determination of the appropriate high-level information to collect was based on questions given in the Software Engineering Institute's *A Guide to the Classification and Assessment of Software Engineering Tools* [Firth 1987], and Brownstein and Lerner's *Guidelines for Evaluating and Selecting Software Packages* [Brownstein 1982]. More detailed information requirements were deduced from the characteristics of the tools themselves.

3. TOOL EXAMINATIONS

This section describes the selected tools in terms of pertinent vendor details, operating environments, and the functionality provided. Price information accurate at the time of examination is also included. Each description is supported with observations that discuss ease of use, documentation and user support, and Ada restrictions. Brief mention of any problems encountered during the examinations provides insight into the reliability and robustness of each tool.

3.1 MALPAS

MALPAS comprises a suite of static analyzers that provide control flow, data use, input/output dependency, and complexity analysis and symbolic execution.

3.1.1 Tool Overview

MALPAS was developed in the late 1970s at the United Kingdom Ministry of Defense Royal Signals and Radar Establishment to verify avionics and other safety-critical defense system software. Since 1986 it has been marketed and supported by Rex, Thompson & Partners (RTP). MALPAS has 50 users, including 5 Ada sites. The Ada translator is a new product released in July 1991. RTP also markets seminars to introduce potential customers to MALPAS and training courses. A users group is supported. MALPAS is available on VAX/VMS platforms. The tools examined in this study were MALPAS Release 5.1, IL Version 5, Pascal-IL Translator 3.1, and Ada-IL Translator 1.01. The price for MALPAS and the Ada-IL translator at the time was $60,000.

The analyses performed by MALPAS are intended to assure software safety, reliability, consistency, and conformance to standards. They include the following:

- Control flow analysis to reveal underlying program structure, unreachable code.
- Data flow analysis to detect uninitialized variables, successive assignments without use.
- Information flow analysis to identify input-output dependencies.
- Path assessment to produce a structural complexity measure.

- Partial analysis using program slicing to reduce analysis time.
- Semantic analysis to provide symbolic execution for each loop-free path.
- Compliance analysis to verify code against formal specifications.

MALPAS analyses are based on an Intermediate Language (IL) representation of program specifications or source code. Translators from several languages (including Ada, C, Fortran, and Pascal) to IL are available. The approach of using a common intermediate language for analyses simplifies the extension of MALPAS's capabilities to other programming languages. Formal program specifications can also be expressed in IL. At present, however, no automated translation tools for other formal specification languages such as OBJ, Vienna Development Method (VDM), or Z are supported.

Analyzing application source code is a two-step process. First the code is translated into IL. Since the Ada translator was not available when the tool examination started, the Pascal translator was examined first. Pascal code is translated as a single complete program; this is a straightforward process. The translation of Ada source code to IL is significantly more complicated. The sample Ada code analyzed contained several separately compiled packages and subunits. First the generic input/output packages used by the program had to be instantiated (by hand), translated, and loaded into an IL code library. Then each program unit had to be translated and loaded into the IL code library.

The second step is to run the analyses on the IL code. A single tool controls all of the available analyses. Options are selected by command line parameters and results are written to files that can be printed. Default parameter settings for initial analyses of new code were set up to include control flow, data use, and information flow analyses. The compliance and semantic analyses are computationally more complex. The partial analysis capability allows these analyses to be restricted to particular modules or paths within the program.

Control flow, data flow, and information flow analyses are fairly standard static analysis techniques. Structured programming has largely eliminated control flow anomalies. Data flow and information flow anomalies, however, are still useful indicators of potential problems. Information flow, for example, identifies all of a subprogram's inputs and outputs, which may be more than those passed as parameters. MALPAS's semantic analysis option provides symbolic execution of loop-free code segments. That is, for each possible path through a segment, the value of each modified variable is given as an algebraic expression in terms of the input variables. This provides valuable feedback to a programmer about the meaning of the code and the results that will be produced

when the code is executed. The compliance testing option uses this same information to check formally specified requirements that have been added to the IL code.

3.1.2 Observations

Ease of use. MALPAS is a batch-oriented tool even though it may be invoked interactively. The only user interaction is through the set of options that can be selected from the command line. The large number of options may make MALPAS "flexible" for expert users. Novice or casual users, however, may have some difficulty controlling non-default processing.

Introducing the intermediate language for analyses may cause problems for some users. All analyses and reports refer to the IL version of the program rather than to the original source code. The mapping back to the original code must be done manually. The intermediate language approach may simplify extending MALPAS to cover a range of different programming languages (by requiring only new IL translators), but it imposes a level of separation between the actual source code and the analyses that must be compensated for by the user.

Translating Ada source code to IL was found to be somewhat more complicated than expected. The sample Ada code analyzed contained several separate packages and subunits, and normally requires several compilation steps. The MALPAS Ada to IL translator, however, required several additional steps that Ada compilers either do not need or are able to hide.

Documentation and user support. Installation and operating instructions were clear, thorough, and accurate. Installation required simply editing sample command files to name local directories and disks. The manuals included good examples and the tools operated exactly as described.

Ada restrictions. Although support for all aspects of Ada that can be analyzed statically is the vendor's eventual goal, the current MALPAS tools support only a subset of Ada. The Ada to IL translator recognizes all valid Ada code but the translation to IL is not complete. The intermediate language, for example, does not include any mechanism for concurrency, so Ada tasks cannot be translated. This restriction is particularly unfortunate because execution-based testing of concurrent programs is often difficult to control. Repeating a particular test, for example, might not produce the same results each time. Rigorous static analyses of potential task interactions would contribute significantly to identifying and correcting tasking problems.

Translation of Ada's generic program units is not supported. Generic units provide a powerful mechanism that simplifies programs and enhances reuse. Ada's standard

input and output facilities, for example, are defined in terms of generic packages. MALPAS currently requires manual instantiation of any required generic units.

Access types (pointers) and dynamic storage allocation are not supported. Analysis of unconstrained use of pointers, for example to detect potential "dangling" pointers, is virtually impossible. A workaround for disciplined use of pointers for data structures such as linked lists is to define abstract data types that encapsulate the pointers. MALPAS would be able to analyze application code that used the abstract data types since the pointers are hidden. MALPAS, however, would not be able to analyze an implementation of the abstraction that used pointers.

Problems encountered. The MALPAS tools performed as specified in their documentation. No failures occurred in use.

3.2 SoftTest

SoftTest supports requirements-based test analysis using cause-effect graphing. It derives test conditions to guide the preparation of test data. It also provides a measure of test adequacy in terms of the number of testable functional variations for which tests have been specified.

3.2.1 Tool Overview

SoftTest was developed in 1987 and is marketed and supported by Bender and Associates. There are currently over 50 users. The tool executes on any IBM PC, XT, AT, PS2 or compatible platform under MS-DOS. Bender also markets project methodology guidelines, consulting services, and training courses on software quality assurance and testing. The version of SoftTest examined was Release 3.1. At the time the price was $2,500.

SoftTest automates a requirements analysis technique called cause-effect graphing, developed at IBM in the early 1970s. The primary phases of analysis are as follows:

- Extraction of node, relation, and constraint definitions from cause-effect graphs.
- Functional variation analysis to identify combinations of input conditions required for tests.
- Test condition synthesis to consolidate variations and produce minimal test sets.

A cause-effect graph identifies the set of input conditions (the causes) that a program must respond to, and relates these to the set of output conditions (the effects) that the program must produce. Relations between inputs, outputs, and intermediate conditions are specified in terms of combinational logic (AND, OR, NOT). Special

relationships such as mutually exclusive input conditions and conditions that hide or mask other conditions can also be specified.

The functional variations of a cause-effect graph reflect all the individual unique functions the software is required to perform. The thesis of this approach to testing is that although the number of possible combinations of input conditions may be very large, a program can be thoroughly tested by exercising this relatively small set of unique functions.

Some functional variations may not be testable because, for example, it may be physically impossible for certain combinations of input conditions to arise. Another reason is because the results of one function may be obscured by other functions. When this latter case arises, SoftTest identifies intermediate program results that, if they could be observed, would enable otherwise obscured functions to be tested.

A single test may exercise several functions. This means that a smaller number of test cases will often be able to exercise all of a program's functionality. SoftTest includes analysis that yields a minimal number of test cases that will exercise all the testable functional variations. Additional tests for special cases such as boundary conditions can be added to the tests produced by SoftTest.

Cause-effect graphs express only combinational relationships, constructed from AND, OR, and NOT operations, between causes and effects. Graphs are not allowed to form loops that connect effects back to causes. This raises a question about how to test programs that clearly require iterative or recursive processing. For example, the functionality of sorting a fixed number of inputs can be specified using AND, OR, and NOT; but sorting arbitrary length lists or files of inputs cannot. By unrolling the first few iterations of the (assumed) loop structure, SoftTest can generate test conditions that achieve virtually any level of coverage. That is, if the functionality required could be implemented by:

```
while not completed loop
   loop_body;
end loop;
```

then the cause-effect graph could be specified as if the implementation would be:

```
if not completed then
   loop_body; -- first iteration
   if not completed then
      loop_body; -- second iteration
      ...
   end if;
end if;
```

The test conditions produced for this specification cover zero, one, and two iterations of the loop body, which would exercise every combination of pairwise linear code segments. Although not exhaustive, this level of coverage is usually considered fairly thorough testing. Additional, "realistic" tests for sorting, for example, could be added to the tests produced by this analysis. Requirements-based testing, of course, is not supposed to use knowledge of a program's structure. The required functionality, however, can strongly indicate an iterative (or recursive) implementation.

3.2.2 Observations

Ease of use. SoftTest's user interface provides simple menu-driven commands to initiate processing and review results. It is also possible to invoke one of a number of third-party text editors from within the tool so that graph specifications can be corrected and analyses rerun without leaving the tool. Analysis reports can also be easily printed. The hard part about using SoftTest is developing complete functional specifications for software to be tested. Even though the cause-effect graph language is clear and simple, writing specifications in this form requires some experience. New users should expect to spend some time with the tutorial materials provided. Training courses offered by the vendor may also prove useful.

Documentation and user support. The tool documentation and user support were quite good. Installation was simple and the tool operated exactly as described in the reference manual. Two tutorials were provided--one that worked through examples of how to run the tool and one that discussed requirements-based testing in more general terms. Bender and Associates answered several questions about cause-effect graphing techniques over the phone.

Ada restrictions. SoftTest is independent of a particular programming language.

Problems encountered. SoftTest performed as documented. No problems were encountered in its use.

3.3 TCAT/Ada, TCAT-PATH, TDGen, S-TCAT/Ada, and TSCOPE

These tools are part of the Software TestWorks toolset that also includes SMARTS, CAPBAK, and EXDIFF for regression testing. Two more tools, SpecTest and MetaTest, are planned for release late 1991 and will use software specifications and designs to guide code testing.

3.3.1 Tool Overview

Software TestWorks has been marketed by Software Research for over five years. Each type of coverage analyzer, that is TCAT, TCAT-PATH, has respectively 2,100,

1,800, and 1,600 users. TDGen has over 400 users, and TSCOPE has 1,100 users. Software Research also offers a range of software testing services, technical seminars, and programming language validation suites. The tools are available on operating platforms ranging from personal computers (PCs) to mainframes under UNIX, MS-DOS, OS-2, and VMS operating systems. TSCOPE requires X-Windows. Prices dependent on the operating environment and start at $1,900 for TCAT/Ada, $1,950 for TCAT-PATH, $800 for S-TCAT/Ada, $300 for TDGen, and $900 for TSCOPE. The examinations were performed on a Sun-4 copy of the TCAT/Ada Version 7.6, S-TCAT/Ada Version 7.6, TCAT-PATH Release 7, TDGen Release 3.2, and TSCOPE Release 1.2.

TCAT/Ada instruments the code contained in a user-specified list of files to reveal whether each module branch has been executed. This process also produces a program listing, called a reference listing, that shows the location and numbering of markers that identify particular code segments. The user then compiles the instrumented program and links it with a provided runtime file. When run, this program queries the user for the name of the tracefile to which execution data will be written. This tracefile is subsequently used by the reporting utility to list the overall coverage achieved, identify hit and not-hit segments, and produce histograms showing the frequency distribution of segments hit using either linear or logarithmic scales. All segment information is given in terms of the segment numbers shown in the reference listing. The reporting utility can combine archived test data with new tracefile data if desired. At the end of each run, the tracefile is combined with the specified archive file to produce a new archive file. With the exception of information on the sequence in which segments were hit, archive files contain the same data as a tracefile. For integration testing, a special utility that monitors the total could-have-been-hit count is provided to prevent initial tests, that may not touch all program modules, from producing artificially high coverage counts. Additional utility programs for abbreviated coverage reports, reporting coverage on individual modules, and analyzing trace files for record types are provided.

S-TCAT/Ada provides the same basic functionality for analyzing module-to-module interfaces that have been exercised as TCAT/Ada provides for branches. Modules are defined to be all Ada procedures and functions, except runtime functions and those appearing in a user-specified list of interfaces. An additional utility is provided to generate a tabular representation of the call graph of the source program.

TCAT-PATH is also similar to TCAT/Ada except that coverage reporting addresses the paths executed and is only provided on a single tracefile; archive files are not supported. In addition to the instrumented code and reference listing, the instrumentor generates a separate digraph for each module. This can be used to count the number of paths and display path statistics. A set of module paths also can be generated from a

digraph with the user limiting path generation or preventing the output of some generated paths as necessary. Additional utilities are available to generate an (approximate) picture, determine the cyclomatic complexity of a digraph, and support use of TCAT-PATH with UNIX-type make files.

TSCOPE can be used with TCAT/Ada, TCAT-PATH, or S-TCAT/Ada to animate test completeness. Each program module can be instrumented for either branch, path coverage, or call-pair coverage. All instrumented modules are reported on a single display and different kinds of reporting can be selected for different modules. Graphic representations such as histograms are available for dynamic displays. Digraphs and call-trees can be displayed either dynamically or statically. TSCOPE employs a graphical user interface and output displays are mapped to an X-Window screen by means of a user-specified configuration file.

TDGen works with two files. The template file tells TDGen how to generate test data based on data supplied in a values file. TDGen replaces variable fields, called descriptors, in the template with values from the values file. Descriptors may be user-defined or take one of the predefined values (ascii, alpha, decimal, and real). When invoking TDGen the user specifies whether values for the descriptors should be taken from integers given in the command line, randomly from the values file, or sequentially from the values file to generate every possible combination of values. To aid this choice, TDGen can be requested to tabulate the number of possible test data combinations. A provision for escaping to the operating system level is provided to allow, for example, editing files during a TDGen session.

3.3.2 Observations

Ease of use. A user can interact with these tools using either a command-line interface or a series of menus. Only the command-line interface versions of the tools were available for the examination reported here. In the case of TCAT-PATH, support for UNIX-like make files is provided to facilitate repetitive compilation and linking tasks. Context-sensitive help and help frames discussing each function are provided. No special knowledge is required to use these tools.

Reports are generally well-structured. Since segments, paths, and call-pairs are referred to by number, however, a user must refer back to the various reference listings to identify the subject of each reference. Path descriptions, given in terms of numbered edges and nodes from the underlying digraph, are particularly difficult to read. The presentation of histograms, digraphs, and call graphs would benefit from the use of modern graphical techniques.

Documentation and user support. The tools are supported by extensive documentation that includes guidelines on appropriate minimum coverage levels. Examination of these tools was delayed by the time it took the vendor to respond to tool problems.

Instrumentation Overhead. TCAT/Ada and TCAT-PATH instrument the contents of files specified as part of the tool invocation. In each case, all code is instrumented the same way. For TCAT/Ada, the vendor recommends a capacity of up to 2500 segments (approximately 20,000 lines of code). The vendor estimates the size/performance overhead for instrumentation at 20% to 30%, although this can be higher for very complex programs. For the Ada Lexical Analyzer Generator, TCAT/Ada, TCAT-PATH, and S-TCAT/Ada instrumentation gave, respectively, 26%, 27%, and 14% increases in code size. The instrumentation introduced by S-TCAT/Ada can be limited by specifying a set of module interfaces that are not to be reported on. Versions of TCAT and S-TCAT that accomplish various levels of in-place buffering to enhance performance are available for C. Similar support is not available for the Ada versions.

Ada restrictions. The coverage analyzers do not instrument overloaded routines or tasking constructs. The TCAT/Ada and S-TCAT/Ada processors (iada and s-iada) have been validated against the Ada validation suite, a set of programs that test compliance with the Ada standard.

Problems encountered. Several problems delayed the examination of these tools. First of all, copies of testing tools for C programs were sent twice before the requested Ada versions finally arrived. Except for TDGen and TSCOPE, only the command-line versions of tools were sent. The menu versions were again requested but never arrived. Initial execution of TCAT/Ada on one of Software Research's example programs caused a segmentation fault and sometimes a core dump. Initial problems encountered with TCAT-PATH and S-TCAT/Ada turned out to be the result of errors in the supplied runtime file. For all three instrumenters, misplaced inserted instrumentation statements had to be manually corrected. In some cases, these misplaced statements caused compilation errors. In other cases, the incorrectly instrumented program would compile but produced the wrong results. Additional problems with accessing help were attributed to poor installation procedures that caused help files to be improperly positioned. In TDGen, errors in values and template file, or in the specification of program options, caused the program to hang.

3.4 TBGEN and TCMON

TBGEN generates testbeds that facilitate unit testing bottom-up integration testing. The next version of this tool will include the generation of stubs so that top-down integration testing is also supported.

3.4.1 Tool Overview

TBGEN and TCMON are marketed by Nokia Data systems. They have been commercially available since 1986, although a fully revised version of TBGEN (Version 3.0) was released in May 1989. 30 permanent multi-user licenses have been sold. Designed to be hardware architecture, operating system, and compiler independent, these tools are available for VAX/VMS, Sun-3/SunOS, PCs under MS-DOS and OS-2, and Rational machines. There are some minor difference between the versions available on different operating environments; for example, unlike the Sun-3/SunOS versions, the VAX/VMS tools do not allow escaping to the operating system command level. TBGEN prices start at $2,850 and TCMON at $2,300. Evaluation copies of the tools, allowing their use for 60 days, are available for $1,000 (one tool) or $1,500 (both tools). The versions examined were TBGEN Version 3.1 and TCMON Version 2.2 operating on a VAX/VMS platform.

Using Ada program unit specifications, TBGEN generates a testbed and a command file for compiling and linking this testbed with the units under test. The user can control the size of a generated testbed by specifying particular subprograms or program objects to be excluded. A log file automatically records pertinent information about testbed generation. The user executes the resulting testbed, specifying the desired calling sequence and subprogram parameters, and observing the results. A powerful set of Ada-like testbed commands is provided. For example, testbed variables can be declared and their visibility directly controlled, and many of the entities declared in Ada specifications can be accessed. Additional commands display information based on current testbed settings and testbed status, or cause user inputs and testbed outputs to be copied to a trace file for later examination. Instead of using a testbed interactively, the user can specify testbed inputs in the form of a script file. Scripts may be user developed or generated in the form of a copy of previous testbed inputs. Conditional and iteration control structures, along with fixed and variable breakpoints, are provide for scripts. Assertions are provided for automatic checking of test results against expected results. Since the testbed takes standard input from the keyboard for interactive communication with the user, some difficulties may be encountered if a module under test also uses standard input.

TCMON instruments the contents of user-selected files with statements that act as measurement probes. In addition to structural coverage, these probes provide for segment execution counts and true/false counts of conditions and subconditions. They also provide timers that allow capturing execution times at the program unit level, and the measurement of times between user-specified events. Each subprogram can be instrumented for different types of monitoring. A test monitor is generated. A command file for compiling the monitor and instrumented code and performing necessary linking is also

generated, together with a log file providing information about instrumented files and units generated. The monitor supports a command-driven interface that provides the user with commands to reset all counters and timers, save and append measurement data, produce a profile listing, run the instrumented program, etc. Where necessary, this interface can be omitted by inserting TCMON commands in source files as special comments and generating a dummy monitor. Data generated by the instrumentation is recorded in a profile listing. This gives detailed information about counter and timer places and values, and a histogram of statement list execution counts is included. The profile listing also contains information that can be used to estimate the influence of instrumentation statements on measured time. The TCMON Postprocessor (TCPOST) processes the profile listing to generate summary reports at either the package or subprogram level.

Timers may include invalid data when two or more tasks call the same instrumented subprogram or are of the same instrumented task type. The same is true for recursive procedures. If this happens, the affected timers are flagged in the profile listing. Although generic procedures and packages can be instrumented, multiple instances are not distinguished. Also, when returning from a function, it is not possible for a timer within the function to record the time spent in the evaluation of the return expression. Exceptions, which are invisible to the instrumentor, may also distort timing results.

3.4.2 Observations

Ease of use. The user interacts with TBGEN and TCMON through command interfaces that are well supported with prompts to guide a user through necessary steps. Context-sensitive help is available, together with general descriptions on user-selected topics. Error messages are informative, though no specific help for resolving an error is provided. They are written to both the display and the appropriate log file. When erroneous input is detected, execution of the current command is terminated and the rest of the current input line ignored. When a test script is being used in TBGEN, processing will continue with the next line. Command files are provided to relieve the user of some repetitive manual labor. Although the use of TCMON requires no special knowledge, the TBGEN command-interface requires a knowledge of Ada. All reports are well-structured and clear, with useful history-keeping information.

TBGEN is tailorable in several ways. The SPECIAL command implements environment or installation specific commands. Configuration parameters specified in a system file can be changed, essentially to modify default file names. A system file gives the specification for package STANDARD which can be modified to reflect some of the options available to Ada compilers. The template files used in generating testbed

components can be changed.

Some aspects of TCMON can also be altered by modifying the template file used for generating auxiliary Ada units and the command file. This template file also contains the configuration parameters that can be changed to alter default values. The TCMON User's Manual provides suggestions for modifying the parent type for counter variables, measuring CPU-time instead of default elapsed time, or including other cost functions.

Documentation and user support. The documentation is well-written and guides a user through using each tool. The vendor provided good support and answered all questions quickly and well.

Instrumentation Overhead. TCMON is designed to minimize the introduction of unnecessary instrumentation. It not only allows the user to select the files whose contents are to be instrumented, but allows each file to be instrumented differently. TCMON also allows the user to select between SAFE or UNCHECKED modes for the segment counter. The vendor cites a 50% to 100% increase in code size for full structural instrumentation. For the Ada Lexical Analyzer Generator, full structural instrumentation of all units gave a size increase of 120%.

Ada restrictions. TBGEN accepts any valid Ada code. Expressions, however, are skipped with the result that the type of an array index cannot always be determined automatically and the user may be asked to supply this information. Tasks, task types, and dependent entities are ignored and cannot be accessed in testbeds directly. Similarly, testbeds do not provide the user with access to objects of limited type, functions with results of limited type, array objects with a constrained array definition, and constrained subtypes of a type with discriminants. TCMON may misinterpret overloaded operators returning boolean values when these are used in conditions.

Problems encountered. No significant problems were encountered during the examinations of these tools.

3.5 TestGen

TestGen is part of the AISLE family of software tools that is based on the Ada-based Design and Documentation Language (ADADL). ADADL itself is fully compilable by any Ada compiler and has been selected by the Joint Integrated Avionics Working Group (JIAWG) as the Ada program design language (PDL) to use for the Army's Light Helicopter Experimental (LHX) and the Air Force's Advanced Tactical Fighter (ATF) programs. In additional to the ADADL processor that provides static analysis of designs, the tool family includes syntax-directed and graphical editors, a design and code quality analyzer, an automatic document generator, and design database analysis tools

and supports requirements traceability. It can interface to the Teamwork, Software Through Pictures, and Excelerator CASE systems [Hook 91] to provide automatic generation of designs from requirements. The examination focused on TestGen which can be applied both to ADADL designs and Ada code.

3.5.1 Tool Overview

The AISLE tool family is marketed by Software Systems Design (SSD). It has been available since 1984 and has over 1,000 users. The tools are available on a wide range of machines such as VAX, VAXStation, and MicroVAX under VMS, UNIX, or Ultrix; and Sun-3 and Sun-4, HP9000-800, Apollo, DecStation, and 80386-based PC systems under MS-DOS or UNIX. Where windowing is required, the tools support X-Windows, SunWindows, DECwindows, Tektronix, and Hewlett-Packard windows. Graphics output is formatted for a range of devices. These formats include Postscript, Tektronix, and Graphical Kernel System (GKS). TestGen prices start at $4,600 and those for ADADL, required for TestGen preprocessing, at $5,000. Training and consulting services are available.

These examinations used Demonstration Version 2.0.3 of the tools operating on a Sun-4 system. The demonstration version costs $150 and is supposedly fully functional with the exception that input files are limited to 450 lines. During the examination of TestGen, however, it was found that the demonstration version only identifies 50% of paths and reports the coverage achieved on only 1 program unit. Additionally, it only supports X-Windows and SunWindows. The restriction on input file size precluded application of some parts of the TestGen on the Ada Lexical Analyzer Generator. In these cases, SSD executed the necessary steps using a full version of the tool and returned the results to the examiner for analysis.

Two parts of TestGen were considered: the Unit Test Strategy Generator and the Test Coverage Analyzer. (The third component, the Design Review Expert Assistant was not examined.) The first of these parts calculates the total number of paths and branches and the cyclomatic complexity for each selected program unit. It also identifies any unexecutable paths and generates control graphs of the code. This information is used in estimating testing costs in terms of the number of test cases required for structural analysis. TestGen supports three structural dynamic analysis techniques: path testing, branch testing, and structured testing based on McCabe's cyclomatic complexity number [McCabe 1976]. Once the user specifies the types of analysis required, the Unit Test Strategy Generator identifies the conditions required at each decision point and the statements executed under those conditions. This information helps the user derive necessary test data for structural testing. The Test Coverage Analyzer is then used to instrument

user-specified program units. It also generates a simple testbed for the main Ada procedure that performs a loop calling the instrumented programs for as long as the user wishes. If a main procedure is not present, a special testbed must be manually created by the user based on a template supplied in the documentation. The user compiles and links the instrumented code and testbed. As the program executes on test data, the instrumentation produces a trace history that records the order in which statements were executed. The Test Coverage Analyzer is then used to analyze this trace history and report on statement and path coverage, and statement and program unit execution profiles. Reporting on the coverage accumulated over a series of test runs is achieved by requiring analysis of multiple trace histories.

3.5.2 Observations

Ease of use. TestGen is menu-driven, requiring input from the keyboard. An on-line manual is provided instead of on-line help. Error messages are terse and only minimal checking of user input is provided. For example, in one case the lack of checking for invalid file names led to a segmentation fault. No special knowledge is required to use the tool. All output reports are well-structured and provide easy-to-read information. A major strength of this tool is the clear identification of the path conditions that guide the execution of particular program paths. Other than setting default values for files names, the user interface cannot be tailored.

Documentation and user support. The installation instructions were not very clear, but other documentation was good. SSD staff provided quick and helpful support. They acted on problem reports immediately, providing resolutions to the encountered problems within a day or even within a couple of hours.

Instrumentation Overhead. The entire program must be analyzed by the ADADL processor before the Unit Test Strategy Generator can be used. Subsequently, TestGen functions can be invoked for the files analyzed by the ADADL processor. The size of instrumented code is minimized by allowing the user to specify which modules in the selected file should be instrumented. All selected modules are instrumented in the same fashion. Instrumentation of the Ada Lexical Analyzer Generator gave a 50% increase in code size. TestGen accumulates the data generated by instrumentation statements in memory; this may limit the amount of code that can be monitored.

Ada restrictions. The Unit Test Strategy Generator can analyze any Ada code, but the Test Coverage Analyzer cannot instrument tasks. Source lines with multiple statements may be instrumented incorrectly.

Problems encountered. TestGen requires that the software under test first be analyzed by the ADADL processor; problems were encountered during the initial use of this processor. In one case, the ADADL processor went into an infinite loop when run on example code provided by SSD. These problems were reported to SSD who subsequently identified one defect in the ADADL processor and two defects in TestGen. Workarounds were provided. Some problems were the result of inadequate documentation or error checking. For example, TestGen does not check whether the specified window format is actually available; if it is not, the tool crashes. It should be remembered that the examination was performed on the demonstration version of TestGen; these problems may not occur with the licensed versions.

4. CONCLUSIONS

An overview of the examined tools is provided by Table 3 which summarizes the tool features previously reported. The first part of this section lists findings concerning the status of the examined tools that may affect their use. The last part comments on identified gaps in testing tool functionality.

4.1 Findings on the Status of Commercial Tools

Findings drawn from the examination of these commercial tools are as follows:

Finding 1: Some of the examined tools could be brought into immediate use to improve the cost-effectiveness of testing for SDI software development efforts. The examined tools provide various ways to improve the consistency of defect detection and reduce testing costs. Those that were found to be robust and easy to use are suitable for immediate use. In particular, TBGEN offers valuable support for automatically generating the test beds needed for both unit and integration testing. TCMON not only provides for ensuring the adequacy of a set of test data in terms of the level of structural coverage required, but is the only examined tool that supports timing analysis. TBGEN and TCMON also provide useful records of testing activities. SoftTest supports the systematic development of test data and measures test data adequacy in terms of functional coverage. TDGen provides systematic generation of large amounts of user-defined test data.

Finding 2: The coverage analyzers provide reporting data that can support the management of SDI testing efforts. Several of the examined tools support test management by reporting on test progress in terms of percentage of the required structural coverage that has been achieved to date. The majority of these tools are intended to measure coverage at the unit testing but base their measurement on different structural features. Although these tools provide module execution counts that provide some support for measurement at the integration level, only S-TCAT/Ada measures coverage achieved during integration testing. The coverage analyzers vary in their ability to estimate the number of test cases needed, to support the development of test data, and to handle archive files of test results. Only TCMON allows the user to specify the required coverage level.

TABLE 3. Summary of Tool Features

Tool Features / Tool Class	Static Analyzer	Test Data Gen.		Test Bed Gen.	Coverage Analyzer					
Tool Name	MALPAS	SoftTest	TDGen	TBGEN	S-TCAT	TCAT	TCAT-PATH	TSCOPE	TCMON	TestGen
Marketed Since	'86	'87	'90	'86	'87	'86	'88	'89	'86	'84
Starting Price $K	60	2.5	0.3	2.85	2.5	1.9	1.95	0.9	2.3	4.6
Range of Platforms	●	●	◎	●	◎	◎	◎	●	●	◎
# Languages Supp.	>5	n/a	n/a	1	4	5	5	n/a	1	2
Related Products	s	s	s,t,v	-	s,t,v	s,t,v	s,t,v	s,t,v	-	s
Vendor Support	○	○	●	○	●	●	●	●	○	○
Ease of Installation	◎	○	○	●	○	○	○	○	●	●
Tool Tailorability	●	●	●	○	-	-	●	-	○	●
Documentation	○	◎	○	◎	◎	◎	◎	○	◎	◎
Absence of Problems	○	○	●	○	●	●	●	◎	○	●
On-Line Help	-	-	-	◎	●	●	●	-	◎	-
Ease of Use	●	○	○	○	●	●	◎	●	○	◎
Test Phases	U,I	S	U,I,S	U,I	I	U	U	U,I	U	U
Activity Logging	○	○	-	○	●	●	-	-	○	-
Tool Dependencies	-	-	-	-	-	-	-	tcat s-tcat tcat-path	-	adadl
Ada Coverage	●	n/a	n/a	○	●	●	●	n/a	○	●

Key to Related Products:
s - seminars/consulting
t - testing services
v - language validation suites

○ ◎ ● ● ●
Better → Worse

Key to Test Phases Supported:
U - unit testing
I - integration testing
S - system testing

Finding 3: Many of the tools can be used in conjunction to overcome the limitations of particular tools. The tools are all intended to provide support for specific, limited testing activities. No single tool provides all desirable functionality. MALPAS, SoftTest, S-TCAT/Ada, TBGEN, and TDGen each provide different functionality. They all operate in an independent manner and can be used in conjunction with each other. TCAT/Ada, TCAT-PATH, TCMON, and TestGen provide similar types of functionality; each can be used with any combination of MALPAS, SoftTest, and TDGen. In addition to its companion tool TCMON, it may be possible to use TBGEN to generate test beds for use with TCAT/Ada and TCAT-PATH.

Finding 4: Only a few of the tools support test data generation, and this support is generally indirect. Only TDGen actually generates test data. This tool operates independently of a functional specification, design, or code. It is independent of any particular test strategy and the effectiveness of generated test data depends on the user. Some other tools, namely SoftTest and TestGen, identify values that certain program variables should take, requiring a user to work backwards to determine the necessary values for program inputs.

Finding 5: The use of some tools may impose restrictions on code development. Only SoftTest and TDGen are independent of code. The other tools are all subject to restrictions on the Ada code they can process. Additionally, TBGEN and all the coverage analyzers receive data on the standard input stream. This presents present difficulties when testing programs that also use the standard input stream. TCMON is the only tool that provides a workaround for this problem. Additional limitations may be found with further use of the tools.

Finding 6: The overhead incurred by code instrumentation needed for structural coverage analysis can be a significant factor. The examined coverage analyzers all limit unnecessary increases in program compilation and execution times by allowing the user to specify files whose contents are to be instrumented. Only TestGen goes beyond this and allows the explicit specification of the particular programs units to instrument. How a tool gathers the large amount of data that instrumentation typically generates is also important. The amount of code that can be monitored may be restricted if this data is accumulated in memory. The alternative approach, writing data to a file, incurs a performance overhead.

Finding 7: The tools do not require special skills but some are immature. The tools

require little sophistication on the part of the user and are supported by good documentation. Some actively guide a user through necessary tasks, keep a record of test activities, and take extra steps to relieve the user of repetitive tasks. In general, however, the tools employ primitive user interfaces that could benefit from the application of human factors engineering. In two cases, MALPAS and TCAT-PATH, the need to refer to separate listings to identify objects referenced in reports complicated tool use. There were instances where different tools gave different results when performing the same function, for example, calculating cyclomatic complexity. Moreover, some of the tools contained faults. While most defects were trivial, others rendered a tool unusable until fixed by the vendor. In three cases, TCAT/Ada, TCAT-PATH, and TestGen, major failures occurred when using the tool on sample software supplied by the vendor.

Finding 8: The examined tools all support mature testing approaches. The functional and structural testing approaches supported by the majority of these tools have been available since the early 1970s. Although the effectiveness of these approaches cannot be directly stated in terms of freedom from defects, there is considerable empirical evidence of their benefits [Rowland 1989, Duran 1984, Girgis 1986, Selby 1986]. This is not to say that they cannot be improved; for example, several test strategies have been developed that offer a compromise between the high cost of path testing and the lack of stringency of branch testing [Woodward 1980, Harrold 1989]. These alternative strategies could be useful employed in coverage analyzers.

More data on the practical costs and benefits of particular tools is needed to determine which tools should be recommended for use, to justify these recommendations, and to encourage prospective users. It could also be used to determine further necessary productization efforts or the porting of tools to new operating platforms. To this end, SDI software development managers should be encouraged to conduct their own examinations of testing tools, perhaps using this report as an initial guideline. Lessons learned in the course of such efforts should be fed back to the tool developers to guide the improvement of existing tools and the development of new tools.

4.2 Functional Gaps in Commercial Testing Tools

Despite the efforts of several researchers, for examples see [Goodenough 1975, Cherniavsky 1988, Morell 1988], there is no commonly agreed formal theory of software testing. In the absence of such a theory, a definitive specification of the required functionality of testing tools cannot be developed. One alternative way of compiling a list of

desired functionality is to look at the new testing techniques that researchers are investigating. This type of approach could result in a list that requires automated support for a large number of techniques such as assertion testing [Luckham 1986], data flow testing [Frankl 1986], flavor analysis [Howden 1987], mutation testing [DeMillo 1988], and software fault tree analysis [Cha 1988].

A more pragmatic approach is to look at testing practices and identify where automated support is needed to alleviate difficulties in current practices. This is the approach adopted here. Based on an assessment of these difficulties, and the capabilities of the set of testing tools identified at the start of this work, some of the most critical gaps in commercial tool functionality are the following:

- Support for reproducible testing of concurrent software. The inherent indeterminism of concurrent programs severely complicates path analysis and means that two executions of the same software with the same inputs can produce different behaviors. This lack of reproducibility handicaps, for example, determining the cause of a failure and retesting a modified program. Concurrent programs can also contain a new class of faults, called synchronization faults. Additional tests are needed to check for existence of these faults.

- Automated oracles that, for stated test inputs, can determine the expected outputs. In dynamic testing, determining the expected program outputs for each set of test data is equally important to generating the test data itself. It is also more difficult and time consuming. Since technology for automated oracles is not, and may never be, available, the development of expected outputs will continue to be a major cost source. A number of alternative approaches have been investigated. These include testing techniques that avoid the need for an oracle [DeMillo 1988] and the use of N-version programming [Avizienis 1985]. The use of symbolic evaluation for providing a partial oracle is perhaps the most promising approach [Richardson 1985].

- Automated assessment of software reliability. Currently, software reliability models are the only way of assessing software reliability. Although the Naval Surface Weapons Center has developed a tool that automates some of the most popular models, there is a need for supported commercial tools. Meanwhile, these models are based on hardware reliability theory and their validity in the software arena has been questioned. Further understanding of software reliability is needed.

Research is being conducted in these areas. As appropriate, the timely development of prototypes should be encouraged to help this technology to mature.

5. FURTHER WORK

In determining the current scope of the study, the examination of several tools or tool classes was postponed. Examination of these tools and tools classes will provide a better understanding of the available commercial tools that support testing of Ada code. This section identifies specific tools and tool classes that should be included to complete this study.

5.1 Additional Static and Dynamic Analysis Tools

During the initial data gathering activities of this work several promising static and dynamic analysis tools were identified that, because of limited resources or other reasons, were not included in the examinations. As previously discussed, examination of the T testing tool has been postponed pending the availability of a significantly revised version due for release in September 1991. The first complete version of AdaQuest is still awaited. Preliminary information on AdaQuest and T is given below. Additional analysis tools that should be examined include the following:

- Static and Dynamic Code Analyzer (ARC SADCA) from Optimization Technology, Inc. The Ada version of this new tool is expected by the end of 1991.
- TST from Intermetrics. A testbed generation tool for Ada subprograms and tasks, this tool is based on the Ada Test Support Tool whose development was funded by the STARS program. Its commercial availability is still uncertain.
- Analysis of Complexity Tool (ACT), Battlemap Analysis Tool (BAT), Design Complexity Tool, and Structure Testing and Requirements Tool (START) from McCabe and Associates. These tools provide complexity analysis, test path and condition generation, integration test generation, and data flow scenario generation. The Ada version of the new Instrumentation Tool is expected to become available by the end of 1991.

5.1.1 Preliminary Information on AdaQuest

AdaQuest is under development for operation on VAX machines. Advertised prices depend on the operating environment and start at $6,500. The dynamic analyzer supports branch testing and verifies unit and subsystem timing requirements by measuring the actual time spent in user-specified code sections. Assertions are used for checking

unit-internal design constraints and interface constraints. In addition to supporting test data preparation for branch testing, the static analyzer checks for logic errors such as infinite loops, unreachable statements, and uninitialized variables. It also checks for violation of project-specific coding standards and generates various dependency reports. Finally, the task analyzer traces the actual synchronization relationships between Ada tasks, creating timing diagrams that may help in diagnosing synchronization errors such as deadlock and starvation.

5.1.2 Preliminary Information on T

T is already used by various government organizations, including the Naval Avionics Center, the Jet Propulsion Laboratories, Naval Coastal Systems Center, and the US Army Forts Monmouth and Sill. It is available for PC/MS-DOS and VAX/VMS platforms, and various workstations under UNIX. Training is a prerequisite for tool purchase and costs $1,500 per person at PEI, or $10,000 for an on-site workshop. Software prices start at $7,000. An interface between T and the Teamwork CASE system is due to be marketed by Cadre Technologies, Inc. in early 1992.

T generates test cases from requirements information. This information includes details on software actions, data, events, conditions, and states and is specified in the T Software Description Language (TSDL). T's goal is generation of the minimum number of traceable test cases that will exercise every operation and each of a set of vendor-defined probable errors at least once. It can be used during any software development stage; during maintenance test cases are generated for software changes only. In addition to test data generation, it provides tracing between tests and defined software actions. From user-specified pass/fail results, T also provides a measure of test adequacy based on requirements coverage, input domain coverage, output range coverage and, optionally, structure coverage.

5.2 Other Classes of Testing Tools

Regression testing tools are widely believed to offer significant cost savings during software development and maintenance. For example, based on its industrial research and testing services, one leading testing company reports that 30% of a maintenance budget is typically spent on retesting software, and automated regression testing can result in savings of 20-25% [SR 1988b]. A sample set of regression testing tools should be examined.

There is increasing evidence that CASE systems are starting to support testing activities. A recent survey of CASE vendors, performed on behalf of the US Air Force, found that nearly 25% of the examined products claim explicit support for software

testing activities [Hook 91]. The goal of integrating testing support into a CASE system is to provide easy access to testing tools that facilitates continual evaluation of evolving software. This evaluation can be used to ensure timely detection of faults and provide the software developer with feedback to guide the development process. The tools examined in this report can be used in conjunction with any CASE system that supports the development of Ada software. Integration into a CASE system, however, requires representing the testing processes and products in the supported life cycle model. A sample set of those CASE systems that offer testing support should be examined to determine how thorough and well-integrated this support is.

REFERENCES

ANSI/MIL-STD-1983 Military Standard MIL/ANSI-STD-1815A. January 1983. *Ada Programming Language.*

Avizienis 1985 — Avizienis, A. December 1985. "The N-Version Approach to Fault-Tolerant Software." *IEEE Transactions on Software Engineering*, Vol. 11, No. 12, pp. 1491-1501.

Boehm 1988 — Boehm, B.W. and P.N. Papaccio. October 1988. "Understanding and Controlling Software Cost." *IEEE Transactions on Software Engineering*, Vol. 14, No. 10, pp. 1462-1477.

Brownstein 1982 — Brownstein, I, and N.B. Lerner. 1982. *Guidelines for Evaluating and Selecting Software Packages.* Elsevier.

Bishop 1986 — Bishop, P.G. et al. August 1986. "PODS - A Project on Diverse Software." *IEEE Transactions on Software Engineering*, Vol. 12, No. 9, pp. 929-940.

Brykczynski 1990 — Brykczynski, B.R., R.N. Meeson, C. Youngblut. 1990. *A Strategic Defense Initiative Organization Software Testing Initiative.* Institute for Defense Analyses, IDA Paper P-2493.

Cha 1988 — Cha, S.S., N.G. Leveson, and T.J. Shimeall. 1988. *Safety Verification in Murphy Using Fault Tree Analysis.* University of California.

Cherniavsky 1988 — Cherniavsky, J.C. 1988. "Testing: An Abstract Approach." In Proc. *ACM SIGSOFT '88 Second Workshop on Software Testing, Analysis, and Verification*, July 19-21, Banff, Canada, pp. 38-44.

DeMillo 1988 — DeMillo, R.A. et al. 1988. "An Extended Overview of the Mothra Software Testing Environment." In *Proceedings 2nd Workshop on Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada.

DefSci 1990 — Aerospace and Defense Science. August 1990. "5th Annual Directory of Ada Tools and Services."

DoDD 1991 — DoD Directive 5000.1. February 23, 1991. *Defense Acquisition.*

Dunn 1984 — Dunn, R.H. 1984. *Software Defect Removal.* McGraw-Hill.

Duran 1984 — Duran. J. and S. Ntafos. July 1984. "An Evaluation of Random Testing." *IEEE Transactions on Software Engineering*, Vol. 10, No. 7,

pp. 438-444.

Fagan 1986 — Fagan, M.E. July 1986. "Advances in Software Inspections." *IEEE: Transactions on Software Engineering*, Vol. SE-12, No. 7, 744-751.

Firth 1987 — Firth, R. et al. August 1987. *A Guide to the Classification and Assessment of Software Engineering Tools*. Software Engineering Institute. Technical Report CMU/SEI-87-TR-10.

Frankl 1986 — Frankl, P.G., and E.J. Weyuker. 1986. "Data Flow Testing in the Presence of Unexecutable Paths." In *Proc. ACM Workshop on Software Testing*, July 15-17, Banff, Canada, pp. 4-13.

Girgis 1986 — Girgis, M.R. and M.R. Woodward. 1986. "An Experimental Comparison of the Error Exposing Ability of Program Testing Criteria." In Proc. *ACM SIGSOFT '86 Symposium on Software Testing, Analysis, and Verification*, 15-17 July, Banff, Canada, pp. 64-73.

Goodenough 1975 — Goodenough, J.B., and S.L Gerhart. June 1975. "Towards a Theory of Test Data Selection." *IEEE Transactions on Software Engineering*, Vol. 1, No. 2, pp. 156-173.

Harrold 1989 — Harrold, M. and M. Soffa. 1989. "Interprocedural Data Flow Testing." In *Proc. ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification*, December 13-15, Key West, Florida, pp. 158-167.

Hook 1991 — Hook, A.A. et al. 1991. *Availability of Ada and C++ Compilers, Tools, Education, and Training*. Institute for Defense Analyses. Institute for Defense Analyses, draft IDA Paper P-2601.

Howden 1987 — Howden, W.E. 1987. *Functional Program Testing and Analysis*. New York: McGraw-Hill.

Luckham 1986 — Luckham, D.C. 1986. *Anna: A Language for Specifying and Debugging Ada Software*. University of Stanford.

Lutz 1990 — Lutz, M. May 1990. "Testing Tools." *IEEE Software*, Vol. 7, No. 3, pp. 53-57.

McCabe 1976 — McCabe, T.J. December 1976. "A Complexity Measure." *IEEE Transactions on Software Engineering*, Vol. 2, No. 4, pp. 308-320.

Meeson 1989 — Meeson, R.N. 1989. *Ada Lexical Analyzer Generator User's Guide*. Institute for Defense Analyses, IDA Paper P-2109.

Morell 1988 — Morell, L.J. "Theoretical Insights into Fault-Based Testing." In Proc. *ACM SIGSOFT '88 Second Workshop on Software Testing, Analysis, and Verification*, July 19-21, Banff, Canada, pp. 45-62.

Myers 1988 — Meyers, W. September 1988. "Shuttle Code Achieves Very Low Error Rate." *IEEE Software*, Vol. 5. No. 6, pp. 93.

Pressman 1987 — Pressman, R.S. 1987. *Software Engineering: A Practitioners Approach*. McGraw-Hill.

Richardson 1985 — Richardson, D.J., and L.A. Clarke. 1985. "Testing Techniques Based on Symbolic Evaluation." In *Software: Requirements, Specification, and Testing*. T. Anderson (ed.), 93-110. Blackwell Scientific Publications.

Rowland 1989 — Rowland, J.H. and Y. Zuyuan. 1989. "Experimental Comparison of Three System Test Strategies." In Proc. *ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification*, December 13-15, Key West, Florida, pp. 141-149.

SDI 1991 — SDIO Directive No. 3405. 1991. *Strategic Defense System (SDS) Software Policy*.

Selby 1986 — Selby, R.W. "Combining Software Testing Strategies: An Empirical Evaluation." In Proc. *ACM SIGSOFT '86 Symposium on Software Testing, Analysis, and Verification*, July 15-17, Banff, Canada, pp. 82-90.

SR 1988a — Software Research Inc. July 1988. *Testing with TCAT: Cost/Benefit Analysis*.

SR 1988b — Software Research Inc. October 1988. *Testing with SMARTS and CapBak: Cost/Benefit Analysis*.

SR 1991 — Software Research, Inc. 1991. *TCAT/Ada Test Coverage Analysis Tool for Ada*. Users Manual UM-1368/2.

Tai 1989 — Tai, K.C., R.H. Carver, and E.E. Obaid. 1989. "Deterministic Execution Debugging of Concurrent Ada Programs." In *Proceedings 13th Annual International Computer Software and Applications Conference*, September 20-22, Orlando, Florida, pp. 102-109.

Weyuker 1988 — Weyuker, E.J. 1988. "An Empirical Study of the Complexity of Data Flow Testing." In *Proceedings 2nd Workshop on Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, pp. 188-195.

Woodward 1980 — Woodward, M.R., D. Hedly, and M. Hennell. May 1980. "Experience with Path Analysis and Testing of Programs." *IEEE Transactions on Software Engineering*, Vol.6, No. 3, pp. 278-286.

Youngblut 1989 — Youngblut, C. et al. 1989. *SDS Software Testing and Evaluation: A Review of the State-of-the-Art in Software Testing and Evaluation with Recommended R&D Tasks*. Institute for Defense Analyses, IDA Paper P-2132.

ACRONYMS

ACM	Association for Computing Machinery
ADADL	Ada-based Design and Documentation Language
AISLE	Ada Integrated Software Lifecycle Environment
ARC SADCA	Static and Dynamic Code Analyzer
ATA	Advanced Tactical Aircraft
ATF	Advanced Tactical Fighter
CASE	Computer-Aided Software Engineering
CPU	Central Processing Unit
DoD	Department of Defense
DoDD	Department of Defense Directive
GKS	Graphical Kernel System
IDA	Institute of Defense Analyses
IEEE	Institute of Electronics and Electrical Engineers
IL	Intermediate Language
JIAWG	Joint Integrated Avionics Working Group
KLOC	Thousand Lines of Code
LHX	Light Helicopter Experimental
MALPAS	Malvern Program Analysis System
MIL-STD	Military Standard
NASA	National Aeronautics and Space Administration
PC	Personal Computer
PDL	Program Design Language
PEI	Programming Environment Institute
SR	Software Research

RTP	Rex, Thomspon and Partners
S-TCAT	System Test Coverage Analysis Tool
SCORE	SDI Cooperative Research Exchange
SDI	Strategic Defense Initiative
SDIO	Strategic Defense Initiative Organization
SDS	Strategic Defense System
SEI	Software Engineering Institute
SIGSOFT	Special Interest Group on Software Engineering
SSD	Software Systems Design
STARS	Software Technology for Adaptable, Reliable Systems
STW	Software TestWorks
TBGEN	Test Bed Generator
TCAT	Test Coverage Analysis Tool
TCAT-PATH	Path Test Coverage Analysis Tool
TCMON	Test Coverage Monitor/Program Bottleneck Finder
TCPOST	TCMON Postprocessor
TDGen	Test Data Generator
TSDL	T Software Description Language
VDM	Vienna Development Method

APPENDIX A

VENDORS OF TESTING TOOLS

TABLE A-1. Vendors of Commercially Available Testing Tools

Vendor	Phone #	Tool Name and Function	Lang.
ABRAXAS Software Inc.	(800) 347-5214	CODE CHECK complexity, portability, style analyzer	C, C++
AETECH, Inc.	(619) 755-1277	AdaScope interfaces symbolic debugger to code	Ada
ACT Corp.	(212) 696-5600	AdaSoft analyzer/debugger	Ada
ASA Inc.	(214) 245-4553	Hindsight logic, complexity, performance, productivity analyzer	C
ATI	(212) 354-8280	superCASE CASE system w' dynamic analyzer	Fortran
Alsys Inc.	(617) 270-0030	AdaTune non-intrusive performance, coverage analyzer	Ada
BTree Verification Systems, Inc.	(612) 474-3756	SVS regression analyzer	n/a
Bender and Associates	(415) 924-9196	SoftTest functional test data generator	n/a
Cadre	(703) 875-8670	SAW non-intrusive performance analyzer XRAY code execution simluator/debugger CodeMap coverage, stack space analyzer SmartProbe debugger w' trace and breakpoints Evaluator regression analyzer	Ada, C n/a
Charles Stark Draper Lab.		DARTS CASE system w' design analyzer	n/a
Clyde Digital Systems, Inc.	(801) 224-5306	CARBONCopy regression analyzer	n/a
Computer Associates	(203) 627-8923	TRAPS regression, acceptance tester	n/a
Concurrent Computer Corp.	(201) 758-7531	XPA performance analyzer Simulation Wbench real-time performance analyzer	n/a n/a

Vendor	Phone #	Tool Name and Function	Lang.
Convex Computer Corp.	(214) 497-4383	CXpa performance analyzer	n/a
DDC-1	(602) 944-1883	DACS compiler, non-intrusive symbolic debugger	Ada
Digital Equipment Corp.	(800) 332-4636	PCA coverage, performance analyzer SCA static analyzer Test Manager regression, test analyzer	n/a n/a n/a
Direct Technology	(800) 992-9979	Automator qa regression and test manager	n/a
Donatech Corp.	(515) 472-7474	RealTime Testware non-intrusive executor w' test management	n/a
Dynamic Research Corp.	(508) 475-9090	AdaMAT quality analyzer	Ada
EVB Software Engineering Inc.	(301) 695-6960	DYN complexity analyzer	Ada
General Research Corp.	(805) 964-7724	AdaQuest static, dynamic analyzer J73AVS static, dynamic analyzers RXVP80 coverage, static analyzer	Ada Jovial Fortran
Gimpel Software	(215) 584-4261	Generic Lint static analyzer	C
Goldbrick Software	(714) 760-9196	Bloodhound regression analyzer	n/a
Harris Corp.	(305) 977-5573	AMS quality analyzer	Ada
Hewlett Packard	(301) 670-4300	HP Branch Validator coverage analyzer	C, C++
ITT Avionics	(201) 284-5030	UATL driver for test execution	Ada
i-Logix	(617) 272-8090	Statemate CASE system w' testing functions	Ada, C
Information Processing Tech. Inc.	(415)494-7500	FORTRAN-lint static analyzer lint-PLUS static analyzer	Fortran C
Inst. for Information Industry	+886 2 737 7187	KangaTool/CEA cause-effect analyzer KangaTool/DPA dynamic program analyzer KangaTool/SPA static program analyzer KangaTool/UTT unit test tool	n/a various various various

Vendor	Phone #	Tool Name and Function	Lang.
Integrated Systems, Inc.	(408) 980-1500	AutoCode CASE system w' testing functions	various
Intermetrics, Inc.	(714) 891-4631	TST Symbolic debugger, test data generator; path, performance, coverage analyzer	Ada
JADE	(804) 744-5849	JADE animator, monitor, debugger for simulation	n/a
Jackson Systems Corp.	(203) 683-1976	APJ Workbench Animator for test data against data/program structures	Ada
MCC	(512) 338-3345	SMDC quality analyzer	Ada
McCabe and Associates	(301) 596-3080	ACT comp!exity, test paths/conditions analyzer BAT test path/condition generator DCT design complexity, integ. test START data flow scenario generator	various various various n/a
Mentor Graphics, Inc.	(714) 660-8080	CASE Station CASE system w' testing functions	C
Mercury Interactive Corp.	(408) 982-0100	Runner series regression tester	n/a
Microtec Research		Xray/DX performance, coverage analyzer for embedded code	n/a
Qual Trak Corp.	(408) 274-8867	DDTs defect tracking system fDDTs field defect tracking system	n/a n/a
Nokia Data	+358-31-237317	TBGEN test bed generator TCMON coverage, performance analyzer	Ada Ada
Pacific-Sierra Research	(213) 820-2200	Flint static analyzer	Fortran
Performance Awareness	(919) 870-8800	preVue series functional, regression, performance tester	n/a
Performance Software	(800) 873-6587	V-Test regression analyzer	n/a
Pilot Research Associates, Inc.	(703) 883-2522	Check*Mate regression analyzer	n/a
Popkin Software	(212) 571-3434	System Architect CASE system w' testing functions	n/a
Programming Environments Inc.	(201) 918-0110	T functional test case generator	n/a
Programming Research	(817) 589-0949	PR:QA quality analyzer	C

Vendor	Phone #	Tool Name and Function	Lang.
RTP Software Ltd.	+0252 711414	MALPAS static analyzer	various
RJO	(301) 731-3600	Auto-G CASE system w' testing functions	n/a
Reasoning Systems	(415) 494-6201	Refine CASE system w' testing functions	C
Howard Rubin Associates, Inc.	(914) 764-4931	RA-METRICS project management analyzer FPXpert complexity analyzer	n/a n/a
SQL Systems International	+44 279-641021	PCMS*ADA dependency analyzer	Ada
Set Labs	(503) 289-4758	UX-METRIC quality analyzer PC-METRIC quality analyzer	various various
Softool Corporation	(805) 683-5777	CAC/2167A DoD-STD-2167A compliance analyzer	
Scandura Intelligent Systems	(215) 664-1207	PRODOC CASE system w' coverage analyzer	
Science Applications Inter. Corp.	(703) 553-6171	MAT maintenance analysis tool Path Analysis Tool coverage analyzer	Fortran Fortran
Software Quality Automation	(800) 228-9922	SQAManager test planner w' test management	n/a
Software Recording, Inc.	(214) 368-1196	AutoTester test planner w' regression testing	n/a
Software Research Inc.	(415) 957-1441	TCAT branch coverage analyzer S-TCAT function call coverage analyzer TCAT-PATH path coverage analyzer T-SCOPE coverage animator TDGen random and sequential test data generator SMARTS regression tester CAPBAK keystroke Capture and Playback System EXDIFF file comparator	various various various n/a n/a n/a n/a n/a
Software Systems Design, Inc.	(714) 625-6147	TestGen complexity, path, coverage analyzer QualGen design, code quality analyzer AIEM database design analyzer	Ada, C Ada, C
Sterling Software	(818) 716-1616	TestPro test planner w' test management	n/a

Vendor	Phone #	Tool Name and Function	Lang.
Syscon Corporation	(619) 296-0085	PEP performance analyzer GRAP animator w' CPU usage, call frequency analyzer	
System Design and Development	(303) 449-3634	DCATS regression analyzer	n/a
Tartan Laboratories, Inc.	(412) 856-3600	Ada Scope analyzer	Ada
Teledyne Brown Engineering	352-8533	ACAT complexity analyzer SMART quality assurance system TAGS/RT CASE system w' testing support	Ada Ada n/a
TeleSoft	(619) 457-2700	Telegen2 development environment w' profiler	Ada
Tiburon Systems, Inc.	(703) 920-2321	FERRET testing workstation	various
Verilog SA	(703) 354-0371	Logiscope complexity, required tests, coverage analyzer AGE CASE system w' r/qs test case generator	various n/a
XA Systems Corp.	(800) 344-9223	PATHVU quality analyzer	Cobol
Xinotech Research	(612)379-3844	Program Composer analyzer	Ada

APPENDIX B

SAMPLE OUTPUTS FROM MALPAS

Due to MALPAS's restrictions on analysis of Ada access types, the lexical analyzer code used as a sample test program could not be thoroughly analyzed. To illustrate the reports that MALPAS produces a simple Pascal program was substituted. This program and the MALPAS analysis reports are shown in the following figures.

```
program  average ( input, output );
   {  This program shares a stream between two consumers by merging the  }
   {  processes and evaluating the result of the second process eagerly. }
   type   resulttype = integer;       { consumer process result type }
          streamelement = integer;    { stream element type }
   var    cons1result: resulttype;    { result returned by consumer #1 }
          cons2result: resulttype;    { result returned by consumer #2 }

   { Stream operations }
   procedure advance (var eos: Boolean; var next: streamelement; more: Boolean);
      const   CR = 13;                { Advance the actual input stream }
      var   ch: char;
      begin
         if  more  then
            if  eof  then
               eos := true
            else begin
               eos := false;
               if  eoln  then begin
                  readln;
                  next := CR
                  end
               else begin
                  read(ch);
                  next := ord(ch)
                  end
               end
      end;

   procedure  consume;                { Consume the input stream as one process }
      var  eos: Boolean;     { (count stream elements and sum stream elements) }
           next: streamelement;
      begin
         cons1result := 0;
         cons2result := 0;
         advance(eos,next,true);
         while  not eos  do begin
            cons1result := cons1result + 1;        { count stream elements }
            cons2result := cons2result + next;     { sum stream elements }
            advance(eos,next,true)
            end;
      end;

begin
   consume;
   writeln('The average of ', cons1result:1,
           ' characters is "', chr(cons2result div cons1result), '".')
end.
```

Figure B-1. Sample Pascal Code Illustrating MALPAS Analyses

```
[1]             TITLE average;
[2]
[3]             [ Pascal to Malpas IL Translator - Release 3.0 ]
[4]
[6]             _INCLUDE/NOLIST "USR:[ADATEST.PASCALIL30]FIXED.PREAMBLE"
*** Including file  USR:[ADATEST.PASCALIL30]FIXED.PREAMBLE;1 ***
*** End of file  USR:[ADATEST.PASCALIL30]FIXED.PREAMBLE;1 ***
[7]             _INCLUDE/NOLIST "USR:[ADATEST.PASCALIL30]TEXT.PREAMBLE"
*** Including file  USR:[ADATEST.PASCALIL30]TEXT.PREAMBLE;1 ***
*** End of file  USR:[ADATEST.PASCALIL30]TEXT.PREAMBLE;1 ***
[8]
[10]            CONST cr : integer = +13;
[11]            CONST lit__1__theaverage : char-array = "The average of ";
[12]            CONST lit__2__characters : char-array = " characters is """;
[13]            CONST lit__3 : char-array = """.";
[14]                  [* result returned by consumer #2 *]
[16]
[17]               [* Stream operations *]
[18]
[20]            PROCSPEC advance(INOUT eos : boolean,
[21]                             INOUT next : integer,
[22]                             IN    more : boolean)
[23]            IMPLICIT ([** IL Global Parameter Section ** ]
[24]                      INOUT input : text);
*** WARNING : no DERIVES list specified for procedure  advance
[25]                  [* Advance the actual input stream *]
[26]
[27]            PROCSPEC consume
[28]            IMPLICIT ([** IL Global Parameter Section ** ]
[29]                      INOUT cons1result, cons2result : integer
[30]                      INOUT input : text);
*** WARNING : no DERIVES list specified for procedure  consume
[31]                  [* Consume the input stream as one process *]
[32]                  [* (count stream elements and sum stream elements) *]
[33]
[35]            MAINSPEC (INOUT input : text
[36]                      INOUT output : text);
[37]
[38]            PROC advance;
[40]            VAR ch: char;
[42]         #1:   IF more THEN
[43]         #3:     IF eof__text(input) THEN
[44]         #5:       eos := true
[45]                 ELSE
[46]         #6:       eos := false;
[47]         #7:       IF eoln__text(input) THEN
[48]         #9:         text__readln(input);
[49]        #10:         next := cr
```

Figure B-2. MALPAS Intermediate Language Translation of Sample

```
[50]                  ELSE
[51]       #11:         text__read__char(input, ch);
[52]       #12:         next := char_pos(ch)
[53]        #8:       ENDIF
[54]        #4:     ENDIF
[55]        #2:   ENDIF
[56]     #STOP:  [SKIP]
[56]      #END: ENDPROC [*advance*]
[57]
[58]           PROC consume;
[60]           VAR eos__6: boolean;
[61]           VAR next__6: integer;
[63]        #1:   cons1result :=  0;
[64]        #2:   cons2result :=  0;
[65]        #3:
[65]             advance(eos__6, next__6, true);
*** WARNING : advance has not been fully specified
[66]        #4:   LOOP [while loop]
[67]        #6:   EXIT [while loop] WHEN NOT( NOT eos__6);
[68]        #7:     cons1result := cons1result +  1;
[69]                       [* count stream elements *]
[70]        #8:     cons2result := cons2result + next__6;
[71]                       [* sum stream elements *]
[72]        #9:
[72]             advance(eos__6, next__6, true)
*** WARNING : advance has not been fully specified
[73]        #5:   ENDLOOP [while loop]
[74]     #STOP:  [SKIP]
[74]      #END: ENDPROC [*consume*]
[75]
[76]           MAIN
[79]           VAR cons1result: integer;
[80]                 [* result returned by consumer #1 *]
[81]           VAR cons2result: integer;
[83]        #1:
[83]             consume();
                      ^
*** WARNING : consume has not been fully specified
[84]        #2:   text__write(output, lit__1__theaverage);
                                        ...
[90]     #STOP:  [SKIP]
[90]      #END: ENDMAIN
[93]           [**** WARNING : WARNINGS IN PASS 1 ... See Listing File ]
[95]           FINISH
*** WARNING : Procedure body for text__get has not been defined
*** WARNING : Procedure body for text__page has not been defined
                                        ...
*** WARNING : Procedure body for text__writeln has not been defined
```

Figure B-2. MALPAS Intermediate Language Translation of Sample(continued)

```
After ONE-ONE, 13 nodes removed.
No nodes with self-loops removed.
       Node id   No of pred.  Succ. nodes
       #START         0          #END
         #END         1
After KASAI (from ONE-ONE), No nodes removed.
After HECHT (from ONE-ONE), No nodes removed.
After HK (from HECHT), No nodes removed.
After TOTAL (from HK), No nodes removed.

Control Flow Summary
====================
The procedure is well structured.
The procedure has no unreachable code and no dynamic halts.
The graph was fully reduced after the following stages:
      ONE-ONE, KASAI, HECHT, HK, TOTAL
The graph was not fully reduced after the following stages:
      None
```

Figure B-3. MALPAS Control Flow Analysis of ADVANCE

```
                                    Key
                                    ===
H = Data read and not subsequently written on some path between the nodes
I = Data read and not previously written on some path between the nodes
A = Data written twice with no intervening read on some path between the nodes
U = Data written and not subsequently read on some path between the nodes
V = Data written and not previously read on some path between the nodes
R = Data read on all paths between the nodes
W = Data written on all paths between the nodes
E = Data read on some path between the nodes
L = Data written on some path between the nodes

After ONE-ONE
   From      To     Data Use Expression
   node     node
#START    #END      H : ch   input   more
                    I : input   more
                    U : eos   input   next
                    V : ch   eos   next
                    R : more
                    E : ch   input   more
                    L : ch   eos   input   next

Summary of Possible Errors
==========================
No errors detected
```

Figure B-4. MALPAS Data Use Analysis of ADVANCE

```
Information Flow
================

After ONE-ONE
   From node #START to node #END

      Identifier          may depend on identifier(s)

      eos                 INs/INOUTs : eos    input    more
                          CONSTANTs  : false    true
      next                INs/INOUTs : input    more    next
                          CONSTANTs  : cr
      input               INs/INOUTs : input    more
      ch                  INs/INOUTs : input    more
                          VARs/OUTs  : ch

Identifier                        may depend on conditional node(s)

eos                                #3        #1
next                               #7        #3        #1
input                              #7        #3        #1
ch                                 #7        #3        #1


Summary of Possible Errors
==========================
No errors detected
```

Figure B-5. MALPAS Information Flow Analysis of ADVANCE

```
Semantic Analysis
=================
After ONE-ONE

>From node : #START
To node   :   #END

IF NOT(more)
THEN MAP
ENDMAP
[-------------------------------------------------------------------------------]
ELSIF more AND eof__text(input)
THEN MAP
   eos := true;
ENDMAP
[-------------------------------------------------------------------------------]
ELSIF more AND eoln__text(input) AND NOT(eof__text(input))
THEN MAP
   eos := false;
   next := 13;
   input := readln__text(input);
ENDMAP
[-------------------------------------------------------------------------------]
ELSIF more AND NOT(eoln__text(input)) AND NOT(eof__text(input))
THEN MAP
   eos := false;
   next := char_pos(read__text__char(input));
   input := skip__text__char(input);
   ch := read__text__char(input);
ENDMAP ENDIF
[-------------------------------------------------------------------------------]
```

Figure B-6. MALPAS Semantic Analysis of ADVANCE

APPENDIX C

SAMPLE OUTPUTS FROM SoftTest

```
/*  Test while-loops corresponding to the code template:  */
  /*  iterate:                */
  /*     while not done do    */
  /*        next_iteration;   */
/*  Test using the following expanded logic:  */
  /*  if done_0 then exit_0       */
  /*  else                        */
  /*    do_first_iteration;       */
  /*    if done_1 then exit_1     */
  /*    else                      */
  /*      do_second_iteration;    */
  /*      if done_2 then exit_2   */
  /*      else ...                */

NODES
  Start    = 'Start iteration process'.
  Done0    = 'Iteration process has completed after zero iterations'.
  Exit0    = 'Exit after zero iterations'.
  DoFirst  = 'Perform first step of iteration'.
  Done1    = 'Iteration process has completed after first iteration'.
  Exit1    = 'Exit after first iteration'.
  DoSecond = 'Perform second step of iteration'.
  Done2    = 'Iteration process has completed after second iteration'.
  Exit2    = 'Exit after second iteration'.
  Exit     = 'Exit from loop' OBS.

CONSTRAINTS
  excl(Exit0, Exit1, Exit2).
  mask(not Start, Done0).
  mask(not DoFirst, Done1).
  mask(not DoSecond, Done2).

RELATIONS
  Exit0    :- Start and Done0.
  DoFirst  :- Start and not Done0.
  Exit1    :- DoFirst and Done1.
  DoSecond :- DoFirst and not Done1.
  Exit2    :- DoSecond and Done2.
  Exit     :- Exit0 or Exit1 or Exit2.
```

Figure C-1. SoftTest Cause-Effect Graph Input

```
Functional Variations for: Test template for iteration schemes

              1. If Start and Done0 then Exit0.
              2. If not Start then not Exit0.
              3. If not Done0 then not Exit0.

              4. If Start and not Done0 then DoFirst.
              5. If not Start then not DoFirst.
              6. If Done0 then not DoFirst.

              7. If DoFirst and Done1 then Exit1.
              8. If not DoFirst then not Exit1.
              9. If not Done1 then not Exit1.

             10. If DoFirst and not Done1 then DoSecond.
             11. If not DoFirst then not DoSecond.
             12. If Done1 then not DoSecond.

             13. If DoSecond and Done2 then Exit2.
             14. If not DoSecond then not Exit2.
             15. If not Done2 then not Exit2.

             16. If Exit0 then Exit.
             17. If Exit1 then Exit.
             18. If Exit2 then Exit.
             19. If not Exit0 and not Exit1 and not Exit2 then not Exit.

Complexity Factors:
     variations / primary causes =  4.8
     variations / (primary causes + primary effects) =  3.8
```

Figure C-2. SoftTest Variation Analysis Output

```
Test Cases/Coverage Matrix for: Test template for iteration schemes

TEST CASE  1:

Cause states:
Start = Start iteration process
Done0 = Iteration process has completed after zero iterations

Effect states:
Exit = Exit from loop

------------------------------------------------------------------------------
TEST CASE  2:

Cause states:
Start = Start iteration process
not Done0 = not Iteration process has completed after zero iterations
Done1 = Iteration process has completed after first iteration

Effect states:
Exit = Exit from loop
------------------------------------------------------------------------------
                                     ...
------------------------------------------------------------------------------
VARIATION COVERAGE by test:
Test                      Variations
----   ---------------------------------------------------------------------
  1    1 16
  2    4 7 17
  3    4 10 13 18
  4    2 19
  5    3 9 15 19

TEST COVERAGE by variation:
 Var                      Tests
----   ---------------------------------------------------------------- --------
  1    1
  2    4
  3    5
  4    2 3
  5    Untestable
  6    Untestable
  7    2
                ...
 19    4 5

Tested variations / testable variations = 100 %
```

Figure C-3. SoftTest Test Synthesis Output

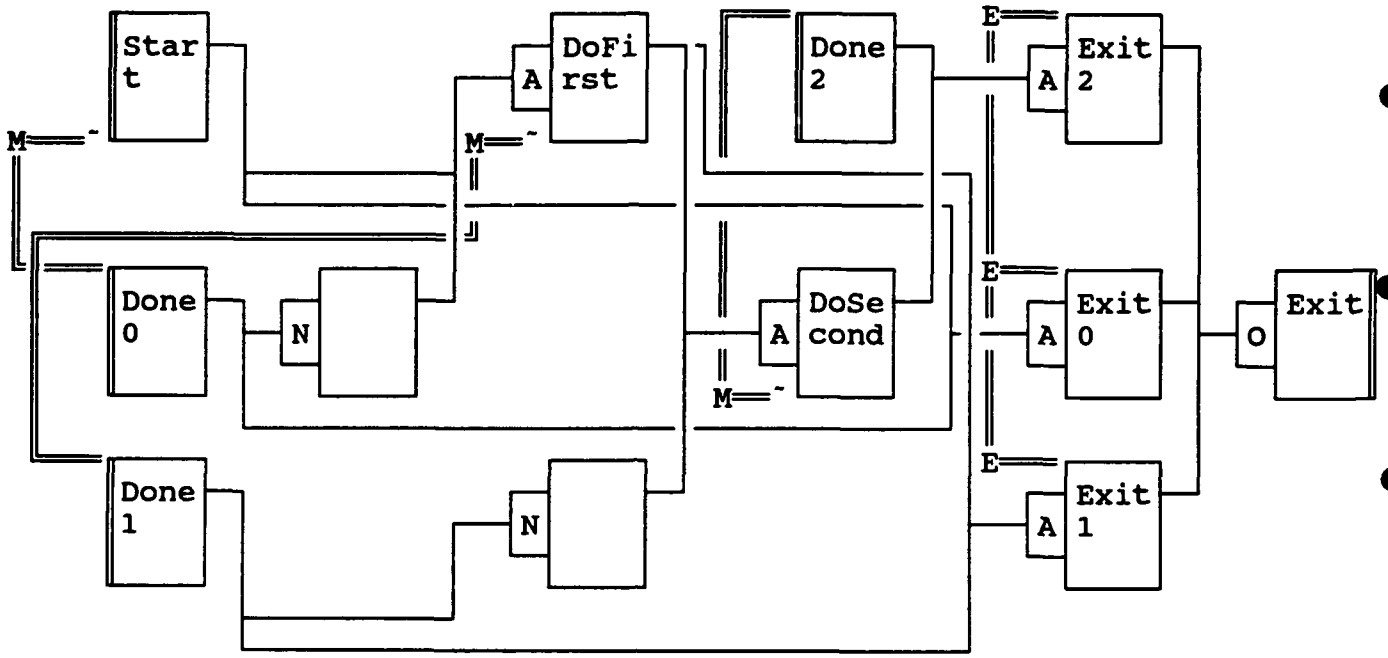


Figure C-4. SoftTest Cause-Effect Graph

APPENDIX D

SAMPLE OUTPUTS FROM TCAT-PATH, TCAT/Ada, and S-TCAT/Ada

```
-------------------------------------------------------------------------
-- TCAT-PATH/Ada, Release 1.2 for UNIX/SUN(tm) (05/05/89)
-- (c) Copyright 1989 by Software Research, Inc.  ALL RIGHTS RESERVED.
-- SEGMENT REFERENCE LISTING Fri Sep  6 12:56:37 1991
--
        procedure  STORE_PATTERN ( PAT_ID, PAT: in LLATTRIBUTE )  is
           --  Store a pattern definition in the pattern table.
           --  Patterns are stored in alphabetical order by name.
        begin
-- START instrumented module "STORE_PATTERN" with 10 segment(s).
-- > STORE_PATTERN/1:
           if  CUR_TABLE_ENTRIES = PATTERN_TABLE_SIZE  then
-- > STORE_PATTERN/2:
-- > STORE_PATTERN/3:
              raise PATTERN_TABLE_FULL;
           end if;
           for  I in 1 .. CUR_TABLE_ENTRIES  loop
-- > STORE_PATTERN/4:
                                  ...
           CUR_TABLE_ENTRIES := CUR_TABLE_ENTRIES + 1;
           PATTERN_TABLE(CUR_TABLE_ENTRIES) := PAT;
           PATTERN_TABLE(CUR_TABLE_ENTRIES).NAME := PAT_ID.STRING_VAL;
        end  STORE_PATTERN;
-- FINISH instrumented module "STORE_PATTERN".

-- END OF TCAT/ADA REFERENCE LISTING
-------------------------------------------------------------------------
-- TCAT/Ada, Release 1.2 for UNIX/SUN(tm) (05/05/89)
-- INSTRUMENTATION STATISTICS
--
-- Module              # segments     # statements  # Conditional statements
-- MERGE_RANGES                 3               0                1
-- ALTERNATE                   17               0                6
-- CHAR_RANGE                   5               0                2
-- RESTRICT                    22               4        825559569
-- TAIL                        17               4                6
-- RESOLVE_AMBIGUITY           20               8           288143
-- COMPLETE_ALT                14               5           288510
                                  ...
-- REPEAT                       4               0        839518514
-- STORE_PATTERN               10               0        839518517
-- ALTERNATE                    1               0           302080
```

Figure D-1. TCAT-PATH Reference Listing

```
-- TCAT-PATH/Ada, Release 1.2 for UNIX/SUN(tm) (05/05/89)
-- (c) Copyright 1989 by Software Research, Inc.  ALL RIGHTS RESERVED.
-- Instrumented module names Fri Sep  6 12:56:37 1991

      MERGE_RANGES       3
      CHAR_RANGE  5
      RESTRICT    22
      TAIL  17
      RESOLVE_AMBIGUITY 20
      COMPLETE_ALT       14
      COMPLETE_CONCAT    13
      COMPLETE_OPT       4
      COMPLETE_PAT       17
      COMPLETE_PATTERNS 3
      CONCATENATE 4
      CVT_ASCII    11
      CVT_STRING  7
      EMIT_ADVANCE_HDR  1
      EMIT_ADVANCE_TLR  1
      EMIT_PKG_DECLS     5
      EMIT_ALT_CASES     7
      EMIT_CONCAT_CASES 8
      EMIT_CONCAT_RIGHT 5
      EMIT_PATTERN_MATCH       32
      EMIT_CHAR    12
      EMIT_SELECT 17
      EMIT_SCAN_PROC     7
      EMIT_TOKEN  18
      INCLUDE_PATTERN    3
      LOOK_AHEAD  1
      LOOK_UP_PATTERN    5
      OPTION       4
      REPEAT       4
      STORE_PATTERN      10
```

Figure D-2. TCAT-PATH Instrumentation Counts for STORE_PATTERN

```
-- TCAT-PATH/Ada, Release 1.2 for UNIX/SUN(tm) (05/05/89)
-- (c) Copyright 1989 by Software Research, Inc.  ALL RIGHTS RESERVED.
-- ERROR LISTING Fri Sep  6 12:56:37 1991

--Module EMIT_PATTERN_NAME overloaded, not instrumented
Internal error loop_edge_back called with no edges to loop back
--Module ALTERNATE overloaded, not instrumented
```

Figure D-3. TCAT-PATH Error Listing for STORE_PATTERN

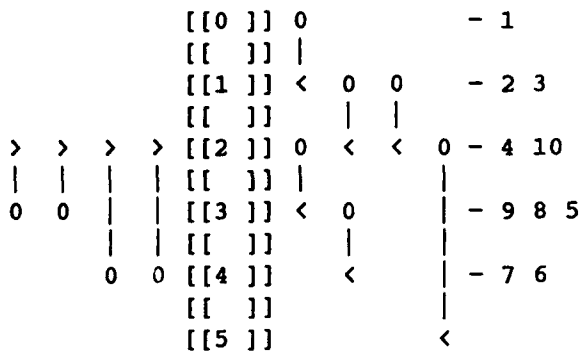


Figure D-4. TCAT-PATH Digraph of STORE_PATTERN

```
 1 2 4 5 6 <[ 4 5 6 7 8 9 ]> 10
 1 2 4 5 7 <[ 4 5 6 7 8 9 ]> 10
 1 2 4 8 <[ 4 5 6 7 8 9 ]> 10
 1 2 4 9 <[ 4 5 6 7 8 9 ]> 10
 1 2 10
 1 3 4 5 6 <[ 4 5 6 7 8 9 ]> 10
 1 3 4 5 7 <[ 4 5 6 7 8 9 ]> 10
 1 3 4 8 <[ 4 5 6 7 8 9 ]> 10
 1 3 4 9 <[ 4 5 6 7 8 9 ]> 10
 1 3 10

Path Analysis Statistics
File name:                      STORE_PATTERN.dig

Number of nodes:                6
Number of edges:                10
Cyclomatic number (E - N + 2):              6

Number of paths:                10
Average path length (segments):             9.80
Minimum length path (segments):             3       (Path 10)
Maximum length path (segments):             12      (Path 7)
Most iteration groups:                  1       (Path 9)

Path count by iteration groups:
      0 iteration group(s):         2
      1 iteration group(s):         8
Stopped at 1 iteration groups
```

Figure D-5. TCAT-PATH Path Analysis of STORE_PATTERN

```
cyclo  [Release 3.3 -- 9/26/90]

Cyclomatic Number  =  Edges - Nodes + 2  =  10 - 6 + 2  =  6
```

Figure D-6. TCAT-PATH Complexity of STORE_PATTERN

```
Ct Test Coverage Analyser Version 1.8
      (c) Copyright 1990 by Software Research, Inc.
Module "STORE_PATTERN": 10 paths, 2 were hit in 5 invocations.
 20.0% Ct coverage

HIT/NOT-HIT REPORT
------------------
P#    Hits    Path text

1     None     1 2 4 5 6 <[ 4 5 6 7 8 9 ]> 10
2     None     1 2 4 5 7 <[ 4 5 6 7 8 9 ]> 10
3     None     1 2 4 8 <[ 4 5 6 7 8 9 ]> 10
4     None     1 2 4 9 <[ 4 5 6 7 8 9 ]> 10
5     None     1 2 10
6     None     1 3 4 5 6 <[ 4 5 6 7 8 9 ]> 10
7     None     1 3 4 5 7 <[ 4 5 6 7 8 9 ]> 10
8     None     1 3 4 8 <[ 4 5 6 7 8 9 ]> 10
9        4     1 3 4 9 <[ 4 5 6 7 8 9 ]> 10
10       1     1 3 10
```

Figure D-7. TCAT-PATH Coverage Report for STORE_PATTERN

```
-------------------------------------------------------------------------
-- TCAT-PATH/Ada, Release 1.2 for UNIX/SUN(tm) (05/05/89)
-- (c) Copyright 1989 by Software Research, Inc.  ALL RIGHTS RESERVED.
-- SEGMENT REFERENCE LISTING Fri Sep  6 12:56:37 1991

        procedure  STORE_PATTERN ( PAT_ID, PAT: in LLATTRIBUTE )  is
           --  Store a pattern definition in the pattern table.
           --  Patterns are stored in alphabetical order by name.
        begin
-- START instrumented module "STORE_PATTERN" with 10 segment(s).

-- > STORE_PATTERN/1:
           if  CUR_TABLE_ENTRIES = PATTERN_TABLE_SIZE  then
-- > STORE_PATTERN/2:
-- > STORE_PATTERN/3:
              raise PATTERN_TABLE_FULL;
           end if;
           for  I in 1 .. CUR_TABLE_ENTRIES  loop
-- > STORE_PATTERN/4:
                                    ...
           CUR_TABLE_ENTRIES := CUR_TABLE_ENTRIES + 1;
           PATTERN_TABLE(CUR_TABLE_ENTRIES) := PAT;
           PATTERN_TABLE(CUR_TABLE_ENTRIES).NAME := PAT_ID.STRING_VAL;
        end  STORE_PATTERN;
-- FINISH instrumented module "STORE_PATTERN".

-- END OF TCAT/ADA REFERENCE LISTING
--
-------------------------------------------------------------------------
-- TCAT/Ada, Release 1.2 for UNIX/SUN(tm) (05/05/89)
-- INSTRUMENTATION STATISTICS
--
-- Module              # segments     # statements  # Conditional statements
--
-- MERGE_RANGES                 3              0              1
-- ALTERNATE                   17              0              6
-- CHAR_RANGE                   5              0              2
-- RESTRICT                    22              4      825559569
-- TAIL                        17              4              6
-- RESOLVE_AMBIGUITY           20              8         288143
-- COMPLETE_ALT                14              5         288510
-- COMPLETE_CONCAT             13              6      839518519
                                    ...
-- REPEAT                       4              0      839518514
-- STORE_PATTERN               10              0      839518517
-- ALTERNATE                    1              0         302080
```

Figure D-8. TCAT/Ada Reference Listing

```
Coverage Analyzer.  [Release 7.3 for SUN/UNIX 12/90]
(c) Copyright 1990 by Software Research, Inc.

Selected COVER System Option Settings:
[-c]  Cumulative Report     -- YES
[-p]  Past History Report   -- YES
[-n]  Not Hit Report        -- YES
[-H]  Hit Report            -- YES
[-nh] Newly Hit Report      -- NO
[-nm] Newly Missed Report   -- NO
[-h]  Histogram Report      -- YES
[-l]  Log Scale Histogram   -- NO
[-Z]  Reference Listing C1  -- NO
```

		(Archived) Past Tests		
No.	Module Name	Number Of Segments:	Number Of Invocations	Number Of Segments Hit	Percent Coverage
0:	BUILDRIGHT	15	128	13	86.67
1:	BUILDSELECT	3	128	3	100.00
2:	CHAR_ADVANCE	4	2	2	50.00
3:	CHAR_RANGE	5	30	5	100.00
		...			
32:	STORE_PATTERN	10	30	5	50.00
	Totals	268	1357	158	58.96

		Current Test			Cumulative Summary		
Module Name:	Number Of Segments:	No. Of Invokes	No. Of Segments Hit	C1% Cover	No. Of Invokes	No. Of Segments Hit	C1% Cover
BUILDRIGHT	15	0	0	0.00	128	13	86.67
BUILDSELECT	3	0	0	0.00	128	3	100.00
CHAR_ADVANCE	4	0	0	0.00	2	2	50.00
COMPLETE_ALT	14	8	9	64.29	28	9	64.29
		...					
TAIL	17	4	4	23.53	4	4	23.53
Totals	341	329	159	46.63	1686	210	61.58

Figure D-9. TCAT/Ada Coverage Report

```
C1 Segment Hit Report.

No.     Module Name:          Segment Coverage Status:

1       BUILDRIGHT            1    2    3    4    5    6    7    8   10
                              11   12   13   15
2       BUILDSELECT           All Segments Hit.  C1 = 100%
3       CHAR_ADVANCE          1    2
4       CHAR_RANGE            All Segments Hit.  C1 = 100%
5       COMPLETE_ALT          1    6    7    8    9   10   12   13   14
6       COMPLETE_CONCAT       1    3    4    5    6   11   13
                              1    2
                                   ...
36      RESTRICT              1    2    6    8    9   10   11   12   17
                              18   19   20   21
37      STORE_PATTERN         1    3    4    9   10
38      TAIL                  1    5   14   16

Number of Segments Hit:          210
Total Number of Segments:              341

C1 Coverage Value:                61.58%


C1 Segment Not Hit Report.

No.     Module Name:          Segment Coverage Status:

1       BUILDRIGHT            9   14
2       BUILDSELECT           All Segments Hit.  C1 = 100%
3       CHAR_ADVANCE          3    4
4       CHAR_RANGE            All Segments Hit.  C1 = 100%
5       COMPLETE_ALT          2    3    4    5   11
6       COMPLETE_CONCAT       2    7    8    9   10   12
                                   ...
36      RESTRICT              3    4    5    7   13   14   15   16   22
37      STORE_PATTERN         2    5    6    7    8
38      TAIL                  2    3    4    6    7    8    9   10   11
                         12   13   15   17

Number of Segments Not Hit:      131
Total Number of Segments:              341

C1 Coverage Value:                61.58%
```

Figure D-9. TCAT/Ada Coverage Report (continued)

```
C1 Segment Hit Report.

No.      Module Name:          Segment Coverage Status:
Segment Level Histogram for Module: BUILDRIGHT

                          +-------------------------------------------------+
                          | Number of Executions, Normalized to Maximum
                          | (Maximum =     348 Hits)   X = One Hit
                          | (Scale:     0.287        Each X =     6.960 Hits)
 Segment  Number Of       |
 Number   Executions      >-1-------20--------40--------60--------80-------100
+-------------------------+-------------------------------------------------+
                          |
    1            128      | XXXXXXXXXXXXXXXXX
    2            348      | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
    3            348      | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
    4            100      | XXXXXXXXXXXXXX
    5            122      | XXXXXXXXXXXXXXXX
    6             92      | XXXXXXXXXXXXX
    7             26      | XXX
    8              8      | X
    9    *                 |
   10            158      | XXXXXXXXXXXXXXXXXXXXX
   11            190      | XXXXXXXXXXXXXXXXXXXXXXXXX
   12            288      | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   13             60      | XXXXXXXX
   14    *                 |
   15            128      | XXXXXXXXXXXXXXXXX
                          |
+-------------------------+-------------------------------------------------+
         (* = Zero Hits)

Average Hits per Executed Segment:  153.5385
C1 Value for this Module:            86.6667
```

Figure D-9. TCAT/Ada Coverage Report (continued)

```
-------------------------------------------------------------------------
-- S-TCAT/Ada, Release 1.2 for UNIX/SUN(tm) (05/05/89)
-- (c) Copyright 1989 by Software Research, Inc.  ALL RIGHTS RESERVED.
-- SEGMENT REFERENCE LISTING Mon Sep  9 13:51:16 1991
--
with  LL_DECLARATIONS, TEXT_IO, INTEGER_TEXT_IO;

package body  LL_SUPPORT  is
                                    ...
         procedure  STORE_PATTERN ( PAT_ID, PAT: in LLATTRIBUTE )  is
            --  Store a pattern definition in the pattern table.
            --  Patterns are stored in alphabetical order by name.
         begin
-- START instrumented module "STORE_PATTERN" with 1 call pairs(s).
            if  CUR_TABLE_ENTRIES = PATTERN_TABLE_SIZE  then
               --  I guess I didn't make the table big enough.
               raise PATTERN_TABLE_FULL;
                                    ...
                  PATTERN_TABLE(I) := ALTERNATE( PAT, PATTERN_TABLE(I) );
-- > STORE_PATTERN/1:
                  PATTERN_TABLE(I).NAME := PAT_ID.STRING_VAL;
                  return;
               end if;
            end loop;
            CUR_TABLE_ENTRIES := CUR_TABLE_ENTRIES + 1;
            PATTERN_TABLE(CUR_TABLE_ENTRIES) := PAT;
            PATTERN_TABLE(CUR_TABLE_ENTRIES).NAME := PAT_ID.STRING_VAL;
         end  STORE_PATTERN;
-- FINISH instrumented module "STORE_PATTERN".

-- END OF S-TCAT/ADA REFERENCE LISTING
-------------------------------------------------------------------------
-- S-TCAT/Ada, Release 1.2 for UNIX/SUN(tm) (05/05/89)
--
-- INSTRUMENTATION STATISTICS
--
-- Module                * segments      # statements
--
-- MERGE_RANGES                   0               0
-- ALTERNATE                      5               0
-- CHAR_RANGE                     0               0
-- RESTRICT                       9               4
-- TAIL                          14               4
-- RESOLVE_AMBIGUITY             28               8
-- COMPLETE_ALT                   3               5
                                    ...
-- STORE_PATTERN                  1               0
-- LL_SUPPORT                     0               0
```

Figure D-10. S-TCAT/Ada Reference Listing

```
------------------------------------------------------------------------
-- S-TCAT/Ada, Release 1.2 for UNIX/SUN(tm) (05/05/89)
-- (c) Copyright 1989 by Software Research, Inc.  ALL RIGHTS RESERVED.
-- Instrumented module names Mon Sep  9 13:51:16 1991

      MERGE_RANGES       0
      ALTERNATE    5
      CHAR_RANGE   0
      RESTRICT     9
      TAIL   14
      RESOLVE_AMBIGUITY 28
      COMPLETE_ALT       3
      COMPLETE_PATTERNS 0
      CONCATENATE 0
                         ...
      REPEAT       0
      STORE_PATTERN      1
      LL_SUPPORT   0
      MERGE_RANGES       0
      ALTERNATE    5
      CHAR_RANGE   0
      RESTRICT     9
      TAIL   14
      RESOLVE_AMBIGUITY 28
      COMPLETE_ALT       3
      COMPLETE_PATTERNS 0
      CONCATENATE 0
      CVT_ASCII    0
      CVT_STRING   1
      EMIT_ADVANCE_HDR   0
      EMIT_ADVANCE_TLR   0
      EMIT_PKG_DECLS     0
      EMIT_ALT_CASES     4
      EMIT_CONCAT_CASES 3
      EMIT_CONCAT_RIGHT 2
      EMIT_PATTERN_MATCH       13
      EMIT_CHAR    0
      EMIT_SCAN_PROC     3
      EMIT_TOKEN   0
      INCLUDE_PATTERN    1
      OPTION       0
      REPEAT       0
      STORE_PATTERN      1
      LL_SUPPORT   0
```

Figure D-11. S-TCAT/Ada Instrumentation Counts

```
ALTERNATE    ALTERNATE
ALTERNATE    ALTERNATE
ALTERNATE    ALTERNATE
ALTERNATE    MERGE_RANGES
ALTERNATE    MERGE_RANGES
RESTRICT     RESTRICT
RESTRICT     RESTRICT
RESTRICT     ALTERNATE
RESTRICT     RESTRICT
RESTRICT     CONCATENATE
RESTRICT     RESTRICT
RESTRICT     OPTION
RESTRICT     OPTION
RESTRICT     CONCATENATE
TAIL  ALTERNATE
TAIL  TAIL
TAIL  TAIL
TAIL  CONCATENATE
TAIL  CONCATENATE
TAIL  ALTERNATE
TAIL  TAIL
TAIL  TAIL
TAIL  CONCATENATE
TAIL  TAIL
TAIL  CONCATENATE
TAIL  TAIL
TAIL  ALTERNATE
TAIL  TAIL
RESOLVE_AMBIGUITY RESTRICT
RESOLVE_AMBIGUITY RESTRICT
RESOLVE_AMBIGUITY ALTERNATE
RESOLVE_AMBIGUITY ALTERNATE
RESOLVE_AMBIGUITY ALTERNATE
RESOLVE_AMBIGUITY ALTERNATE
RESOLVE_AMBIGUITY TAIL
RESOLVE_AMBIGUITY TAIL
RESOLVE_AMBIGUITY OPTION
RESOLVE_AMBIGUITY CONCATENATE
RESOLVE_AMBIGUITY OPTION
RESOLVE_AMBIGUITY CONCATENATE
RESOLVE_AMBIGUITY CONCATENATE
RESOLVE_AMBIGUITY ALTERNATE
RESOLVE_AMBIGUITY CONCATENATE
RESOLVE_AMBIGUITY ALTERNATE
RESOLVE_AMBIGUITY CONCATENATE
                      . . .
STORE_PATTERN     ALTERNATE
```

Figure D-12. S-TCAT/Ada Instrumented Call-Pairs

```
-------------------------------------------------------------------------
-- S-TCAT/Ada, Release 1.2 for UNIX/SUN(tm) (05/05/89)
-- (c) Copyright 1989 by Software Research, Inc.  ALL RIGHTS RESERVED.
-- ERROR LISTING Mon Sep  9 13:51:16 1991

EMIT_PATTERN_NAME not instrumented because overloaded
COMPLETE_PAT not instrumented because overloaded
COMPLETE_CONCAT not instrumented because overloaded
COMPLETE_OPT not instrumented because overloaded
EMIT_SELECT not instrumented because overloaded
LOOK_UP_PATTERN not instrumented because overloaded
--Module EMIT_PATTERN_NAME overloaded, not instrumented
LOOK_AHEAD not instrumented because overloaded
```

Figure D-13. S-TCAT/Ada Error Listing

```
cg 1.12

Root node is ALTERNATE

( 1)  ALTERNATE
( 2)  .   ALTERNATE (* recursive cycle *)
( 2)  .   ALTERNATE (* recursive cycle *)
( 2)  .   ALTERNATE (* recursive cycle *)
( 2)  .   MERGE_RANGES
( 2)  .   MERGE_RANGES

Graph contains 3 cycles.

Disconnected nodes:

    RESTRICT
    CONCATENATE
    OPTION
    TAIL
    RESOLVE_AMBIGUITY
    COMPLETE_ALT
    CVT_STRING
    EMIT_ALT_CASES
    EMIT_PATTERN_MATCH
    EMIT_CONCAT_CASES
    EMIT_CONCAT_RIGHT
    EMIT_SCAN_PROC
    INCLUDE_PATTERN
    STORE_PATTERN
```

Figure D-14. S-TCAT/Ada Control Graph

```
Coverage Analyzer.  [Release 7.3 for SUN/UNIX 12/90]
(c) Copyright 1990 by Software Research, Inc.

Selected SCOVER System Option Settings:
[-c]  Cumulative Report     -- NO
[-p]  Past History Report   -- YES
[-n]  Not Hit Report        -- YES
[-H]  Hit Report            -- NO
[-nh] Newly Hit Report      -- NO
[-nm] Newly Missed Report   -- NO
[-h]  Histogram Report      -- YES
[-l]  Log Scale Histogram   -- NO
[-Z]  Reference Listing S1  -- NO

Options read:   3
S-TCAT: Coverage Analyzer.  [Release 7.3 for SUN/UNIX 12/90]
(c) Copyright 1990 by Software Research, Inc.
```

		(Archived) Past Tests		
			Number Of	
No.	Module Name	Number Of Call-pairs:	Number Of Invocations	Call-pairs Hit	Percent Coverage
0:	EMIT_TOKEN	0	386	0	100.00
1:	CHAR_RANGE	0	20	0	100.00
2:	CONCATENATE	0	112	0	100.00
3:	ALTERNATE	5	68	0	0.00
4:	STORE_PATTERN	1	23	0	0.00
5:	OPTION	0	12	0	100.00
6:	COMPLETE_PATTERNS	0	6	0	100.00
7:	EMIT_PKG_DECLS	0	6	0	100.00
8:	EMIT_ADVANCE_HDR	0	6	0	100.00
9:	INCLUDE_PATTERN	1	17	1	100.00
10:	EMIT_ADVANCE_TLR	0	6	0	100.00
11:	EMIT_SCAN_PROC	3	6	1	33.33
12:	EMIT_CHAR	0	77	0	100.00
			...		
17:	REPEAT	0	4	0	100.00
18:	COMPLETE_ALT	3	26	1	33.33
19:	RESOLVE_AMBIGUITY	28	6	1	3.57
20:	RESTRICT	9	60	1	11.11
21:	TAIL	14	12	0	0.00
22:	EMIT_ALT_CASES	4	6	1	25.00
	Totals	87	996	10	11.49

Figure D-15. S-TCAT/Ada Coverage Report

```
S-TCAT: Coverage Analyzer.  [Release 7.3 for SUN/UNIX 12/90]
(c) Copyright 1990 by Software Research, Inc.

S1 Not Hit Report.
No.    Module Name:           Call-pair Coverage Status:
1       EMIT_TOKEN            All Call-pairs Hit.  S1 = 100%
2       CHAR_RANGE            All Call-pairs Hit.  S1 = 100%
3       CONCATENATE           All Call-pairs Hit.  S1 = 100%
4       ALTERNATE             1    2    3    4    5
5       STORE_PATTERN         1
6       OPTION                All Call-pairs Hit.  S1 = 100%
7       COMPLETE_PATTERNS     All Call-pairs Hit.  S1 = 100%
8       EMIT_PKG_DECLS        All Call-pairs Hit.  S1 = 100%
9       EMIT_ADVANCE_HDR      All Call-pairs Hit.  S1 = 100%
10      INCLUDE_PATTERN       All Call-pairs Hit.  S1 = 100%
11      EMIT_ADVANCE_TLR      All Call-pairs Hit.  S1 = 100%
12      EMIT_SCAN_PROC        2    3
13      EMIT_CHAR             All Call-pairs Hit.  S1 = 100%
14      EMIT_PATTERN_MATCH    2    3    4    5    6    7    8    9   10
                              11   12   13
15      EMIT_CONCAT_RIGHT     2
16      EMIT_CONCAT_CASES     2    3
17      CVT_STRING            All Call-pairs Hit.  S1 = 100%
18      REPEAT                All Call-pairs Hit.  S1 = 100%
19      COMPLETE_ALT          2    3
20      RESOLVE_AMBIGUITY     2    3    4    5    6    7    8    9   10
                              11   12   13   14   15   16   17   18   19
                              20   21   22   23   24   25   26   27   28
21      RESTRICT              2    3    4    5    6    7    8    9
22      TAIL                  1    2    3    4    5    6    7    8    9
                              10   11   12   13   14
23      EMIT_ALT_CASES        2    3    4
Number of Call-pairs Not Hit:          77
Total Number of Call-pairs:            87
S1 Coverage Value:                     11.49%
                              ...
S-TCAT: Coverage Analyzer.  [Release 7.3 for SUN/UNIX 12/90]
(c) Copyright 1990 by Software Research, Inc.
Call-pair Level Histogram for Module: ALTERNATE

                                  No segments hit
S-TCAT: Coverage Analyzer.  [Release 7.3 for SUN/UNIX 12/90]
(c) Copyright 1990 by Software Research, Inc.
Call-pair Level Histogram for Module: STORE_PATTERN

                                  No segments hit
                              ...
```

Figure D-15. S-TCAT/Ada Coverage Report (continued)

```
S-TCAT: Coverage Analyzer.  [Release 7.3 for SUN/UNIX 12/90]
S-TCAT: Coverage Analyzer.  [Release 7.3 for SUN/UNIX 12/90]
(c) Copyright 1990 by Software Research, Inc.
Call-pair Level Histogram for Module: INCLUDE_PATTERN

                        +-------------------------------------------------+
                        | Number of Executions, Normalized to Maximum
                        | (Maximum =     20 Hits)   X = One Hit
                        | (Scale:    5.000       Each X =    0.400 Hits)
 Call-pair Number Of    |
 Number    Executions >-1-------20--------40--------60--------80------100
+---------------------+-------------------------------------------------+
                        |
    1             20    | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                        |
+---------------------+-------------------------------------------------+

Average Hits per Executed Call-pair:  20.0000
S1 Value for this Module:            100.0000
                                ...

S-TCAT: Coverage Analyzer.  [Release 7.3 for SUN/UNIX 12/90]
(c) Copyright 1990 by Software Research, Inc.
Call-pair Level Histogram for Module: EMIT_ALT_CASES

                        +-------------------------------------------------+
                        | Number of Executions, Normalized to Maximum
                        | (Maximum =     12 Hits)   X = One Hit
                        | (Scale:    8.333       Each X =    0.240 Hits)
 Call-pair Number Of    |
 Number    Executions >-1-------20--------40--------60--------80------100
+---------------------+-------------------------------------------------+
                        |
    1             12    | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
    2    *              |
    3    *              |
    4    *              |
                        |
+---------------------+-------------------------------------------------+
        (* = Zero Hits)

Average Hits per Executed Call-pair:  12.0000
S1 Value for this Module:            25.0000
```

Figure D-15. S-TCAT/Ada Coverage Report (continued)

```
{%c Values file 1: for variable number of initial TDGen executions. }

expr         {% expr}{% op}{% expr}  {% identifier}  {% real_no}
op           + - / *
identifier   variable1 variable2 {% alpha 6}
real_no      {% real 4.6} {% integ1}E+{% integ2} {% integ1}E-{% integ2}
integ1       {%r 1..100}
integ2       {%r 3..6}


{%c Values file 2: for last two executions of TDGen. }

expr         variable1 variable2 {% alpha 6}              {% real 4.6} {% integ1}E+{%
op           + - / *
identifier   variable1 variable2 {% alpha 6}
real_no      {% real 4.6} {% integ1}E+{% integ2} {% integ1}E-{% integ2}
integ1       {%r 1..100}
integ2       {%r 3..6}


{%c Template file: Produces arithmetic expression of varying    }
{%c complexity for use in testing a generated lexical analyzer. }

{% expr }
```

Figure D-16. TDGen Sample Value and Template Files

Field	No. Table Entries	Cumulative Total Combinations
% expr	3	3
% op	4	12
% identifier	3	36
% real_no	3	108
% integ1	100	10800
% integ2	4	43200

Figure D-17. TDGen Table of Sequential Combinations for Initial Files

```
{% real_no}
{% expr}{% op}{% expr}
{% identifier}
{% expr}{% op}{% expr}
{% identifier}
{% identifier}
{% expr}{% op}{% expr}
{% real_no}
{% real_no}
{% identifier}
```

Figure D-18. TDGen Output of First Random Execution

```
3E+6
{% real 4.6}-variable1
RSBEz4
{% integ1}E-{% integ2}-{% integ1}E-{% integ2}
variable2
variable2
{% identifier}*{% real_no}/{% identifier}/{% identifier}
21E+4
47E-6
variable1
```

Figure D-19. TDGen Output After 3 Executions with 1st Value File

```
3E+6
3092.703258-variable1
RSBEz4
53E-4-83E-6
variable2
variable2
G36dk5*26E-5/c1mHEJ/variable2
21E+4
47E-6
variable1
```

Figure D-20. TDGen Output After 2 Executions with 2nd Value File

APPENDIX E

SAMPLE OUTPUTS FROM TBGEN AND TCMON

```
-- Script file : USR:[ADATEST.TBGEN]CALENDAR.REC;1
-- Created at  : 1991-08-15 10:37:14
-- Created by  : Test bed CAL_BED generated at 1991-08-15 09:00:03
SET TRACE FILE calendar.trc
DECLARE
  USE calendar
  moment        : time := clock
  current_year  : year
  current_month : month
  the_day       : day_num
  seconds       : day_dura
BEGIN
  split(moment, current_year, current_month, the_day, seconds)

  moment := time_of(current_year, current_month, 15, 0.0)
  DISPLAY day(moment)
  moment := add__1(moment, 86400.0)  -- add__1 equiv to "+"
  split(moment, current_year, current_month, the_day, seconds)
  ASSERT the_day = 16 AND seconds = 0.0

  now     : time := clock
  later   : time := clock
  ASSERT le__1(now, later) = true  -- le__1 equiv to "<="

  moment := time_of(1991, 2, 28, 0.0)
  ASSERT NOT EXCEPTION

  moment := time_of(1991, 2, 29, 0.0)
  ASSERT EXCEPTION(time_error)
END
SET TRACE CLOSED
SET RECORD CLOSED
```

Figure E-1. TBGEN Record File

```
*******************************************************************************
*                        Softplan (R) Ada Tools                               *
*     TBGEN System Version 3.1, Copyright (C) 1990 Nokia Data Systems         *
*                        Test Bed Trace Listing                               *
*******************************************************************************
Test bed generated at 1991-08-15 09:00:03. Time is now 1991-08-15 10:37:22
CAL_BED> DECLARE
CAL_BED>   USE calendar
CAL_BED>   moment        : time := clock
CAL_BED>   current_year  : year
CAL_BED>   current_month : month
CAL_BED>   the_day       : day_num
CAL_BED>   seconds       : day_dura
CAL_BED> BEGIN
CAL_BED>   split(moment, current_year, current_month, the_day, seconds)
         YEAR (out) =  1991
         MONTH (out) =  8
         DAY (out) =  15
         SECONDS (out) =  38266.8500
CAL_BED>
CAL_BED>   moment := time_of(current_year, current_month, 15, 0.0)
CAL_BED>   DISPLAY day(moment)
          15
CAL_BED>   moment := add__1(moment, 86400.0)  -- add__1 equiv to "+"
CAL_BED>   split(moment, current_year, current_month, the_day, seconds)
         YEAR (out) =  1991
         MONTH (out) =  8
         DAY (out) =  16
         SECONDS (out) =  0.0000
CAL_BED>   ASSERT the_day = 16 AND seconds = 0.0
CAL_BED>
CAL_BED>   now     : time := clock
CAL_BED>   later   : time := clock
CAL_BED>   ASSERT le__1(now, later) = true  -- le__1 equiv to "<="
CAL_BED>
CAL_BED>   moment := time_of(1991, 2, 28, 0.0)
CAL_BED>   ASSERT NOT EXCEPTION
CAL_BED>
CAL_BED>   moment := time_of(1991, 2, 29, 0.0)
         *** exception CALENDAR.TIME_ERROR
CAL_BED>   ASSERT EXCEPTION(time_error)
CAL_BED> END
CAL_BED> SET TRACE CLOSED

         Trace closed at 1991-08-15 10:43:46
```

Figure E-2. TBGEN Trace File

```
*******************************************************************************
*                          Softplan (R) Ada Tools                             *
*     TBGEN System Version 3.1, Copyright (C) 1990 Nokia Data Systems         *
*                       Test Bed Generation Log File                          *
*******************************************************************************
Licence identification of the generator:
                    ...
Test bed timestamp...: 1991-08-15 09:00:03

Test bed name.......: CAL_BED
Generated Ada files..: cal*.ada
Command file........: calCMD.COM

Analysed source files:
File: TBGENSYS.STD
File: calendar.spe
                    ...
===============================================================================
The symbol table :

package STANDARD/8001/ is
   type BOOLEAN/1/ is (
      FALSE,
      TRUE);
   type INTEGER/2/ is Integer_Type;
   type FLOAT/3/NoV/ is Float_Type;
   type CHARACTER/4/NoV/ is (
      <>);
   subtype NATURAL/5/NoV/ is INTEGER <2> ;
   -- Type Class  => Integer_Type
   subtype POSITIVE/6/NoV/ is INTEGER <2> ;
   -- Type Class  => Integer_Type
                    ...
      function ">="/2015/(
         LEFT : in     CALENDAR.TIME <11> ;
         RIGHT : in     CALENDAR.TIME <11> )
         return BOOLEAN <1> ;
         -- Alias Name => GE__1
      TIME_ERROR/6006/ : exception;

   end CALENDAR;
end STANDARD;
===============================================================================
Execution of the generator successfully ended at 1991-08-15 09:01:02
```

Figure E-3. TBGEN Generated Log File

```
*****************************************************************
*  TCMON System Version 2.2, (C) Copyright 1987 by Softplan     *
*  Test Coverage Monitor / Program Bottleneck Finder            *
*  Execution profile listing                                    *
*****************************************************************
                                          Counters              Timers
                                         ==========    ========================
LINE    EXECUTION  LE-  PLACE           START/  END/        AVERAGE        TOTAL
NO.     COUNTS     VEL  DESCRIPTION     TRUE    FALSE       TIME           TIME

Source file => [-.adalex2]ll_decls.ada                  Instr => (A,N,N,N)
Source file => ll_compile_dummy.ada                     Instr => (A,N,N,Y)
  23............. 1 proc LL_COMPILE
 120............. 2 func LLFIND
 124*****>>>>>    2 begin              198      0 ?
 127*****>>>>>>>  3 while_loop         898    763 ?
                     Condition         898     63
                                  ...
Source file => [-.adalex2]ll_sup_spec.ada               Instr => (A,N,N,N)
Source file => ll_sup_body_mt.ada                       Instr => (A,Y,Y,Y)
  50............. 1 pack LL_SUPPORT                0                      0.0000
  97............. 2 func ALTERNATE                 14         0.0000       0.0000
 170............. 2 func CHAR_RANGE                6         0.0013       0.0078
 175*****>        2 begin                  6       0 ?
 176                TIMER_CHAR_RANGE              6         0.0013       0.0078
 178***           3 if_then                3       3
                     Condition             3       3
 181***           3 if_else                3       3
 183*****>>>>     4 for_loop              62      62
 188*****>        3 return                 6       0
 204............. 3 proc COMPLETE_ALT              4         0.0039       0.0156
                                  ...
Median of nonzero counter values =  8
One asterisk ( * )              <=> 1
Number of counters                      = 539
Number of timers                        = 36
Number of instrumented (sub)conditions = 206
Minimum measurable time interval        =       0.0001
Estimated cost of one timer operation   =       0.0006
Estimated cost of one counter operation =       0.0000
The instrumentation was done 1991-08-14 13:15:27
This listing was produced    1991-08-14 13:32:48

Data files:
    NAME =>sampleTIM.dat
                  ...
```

Figure E-4. TCMON Profile Execution Listing

```
************************************************************
*  TCMON System Version 2.2, (C) Copyright 1987 by Softplan  *
*  Test Coverage Monitor / Program Bottleneck Finder         *
*  Log of TCMON preprocessor execution                       *
************************************************************
Date and time     => 1991-08-14 13:15:27

Prefix            => SAMPLE
Generated files   => sample*.ada
Main procedure    => *not specified*
Code pattern file => PATTERNS.TCM
                     ...
Source => ll_sup_body_mt.ada
Target => sample_ll_sup_body_mt.ada
Instrumentation => ( INC_MODE        => UNCHECKED
                     COUNTERS        => ALL
                     AUTO_TIMERS     => YES
                     MANUAL_TIMERS   => YES
                     EXPAND_COMMANDS => YES )

  package body LL_SUPPORT on line 50
     function ALTERNATE on line 97
        function MERGE_RANGES on line 103
     function CHAR_RANGE on line 170
     --&& start timer_char_range on line 176 expanded
     --&& stop timer_char_range on line 187 expanded
     procedure COMPLETE_PAT on line 192
                  ...
          Summary Information
          *******************

Number of instrumented files      =     6
Number of compilation units       =     4
Number of body stubs              =     2
Number of subunits                =     2
Number of statement list counters =   539
Number of (sub)condition counters =   206
Number of timers                  =    36
Number of manual timer STARTs     =     2   ... all expanded
Number of manual timer STOPs      =     2   ... all expanded
Number of embedded commands       =     1   ... all expanded

Command file for compilation and linking => SAMPLECMD.COM

ERRORS:    0  WARNINGS:    0
```

Figure E-5. TCMON Log File

```
*****************************************************************************
*  TCMON System Version 2.2, (C) Copyright 1987 by Softplan                 *
*  Test Coverage Summary Report                                             *
*****************************************************************************
PROGRAM                                  STM      COND     SUB      OVER
UNIT                                     LIST     CVRG     COND     ALL
                                         CVRG              CVRG     CVRG

Source file => ll_compile_dummy.ada                        Instr => (A,N,N,Y)
proc LL_COMPILE
 func LLFIND                              88 -     88 -     88 -     88 -
 proc LLPRTSTRING                          0 -      0 -      0 -      0 -
 proc LLPRTTOKEN                           0 -      0 -      0 -      0 -
 proc LLSKIPTOKEN                          0 -                        0 -
 proc LLSKIPNODE                           0 -                        0 -
 proc LLSKIPBOTH                           0 -                        0 -
 proc LLFATAL                              0 -                        0 -
 proc GET_CHARACTER                        0 -      0 -      0 -      0 -
 func MAKE_TOKEN                           0 -      0 -      0 -      0 -
 proc LLNEXTTOKEN                        100      100      100      100
 proc LLMAIN                              62 -     47 -     47 -     56 -
 body LL_COMPILE                         100                        100
-----------------------------------------------------------------------------
proc LL_COMPILE                           49 -     44 -     45 -     48 -
-----------------------------------------------------------------------------
Source file => [-.adalex2]ll_tokens.ada                    Instr => (A,N,N,N)

pack LL_TOKENS
 proc ADVANCE                             83 -     71 -     74 -     79 -
-----------------------------------------------------------------------------
pack LL_TOKENS                            83 -     71 -     74 -     79 -
-----------------------------------------------------------------------------

                  ...
=============================================================================
OVERALL SUMMARY :                         46 -     44 -     44 -     46 -
=============================================================================
Number of partially instrumented or dropped compilation units :  0

This summary was generated at 1991-08-14 13:34:48, based on the TCMON
execution profile listing file sample_out.dat.
The profile listing was produced at 1991-08-14 13:32:48, and the actual
TCPRE instrumentation was performed at 1991-08-14 13:15:27.

There were 103 places out of 128 where the coverage percentage was
below the selected warning level 100 %.
```

Figure E-6. TCPOST Coverage Summary

APPENDIX F

SAMPLE OUTPUTS FROM TestGen

TABLE F-1. TestGen Cyclomatic Complexity Report

page 1 of 3

Module Name	Design	Code
Ll_Support	1	1
Alternate	1	11
Merge_Ranges	1	2
Char_Range	1	3
Complete_Pat	1	12
Complete_Alt	1	8
Restrict	1	14
Tail	1	11
Resolve_Ambiguity	1	13
Complete_Concat	1	8
Complete_Opt	1	3
Complete_Patterns	1	2
Concatenate	1	3
Cvt_Ascii	1	10
Cvt_String	1	4

TABLE F-2. TestGen Test Case Effort Report

page 1 of 3

Number of Test Cases Required for

Module Name	Basis Testing	Branch Testing	Full Path Testing
Ll_Support	1	1	1
Alternate	11	7	27
Merge_Ranges	1	1	1
Char_Range	2	2	2
Complete_Pat	11	11	11
Complete_Alt	7	6	8
Restrict	13	13	13
Tail	8	8	8
Resolve_Ambiguity	12	9	36
Complete_Concat	6	6	6
Complete_Opt	3	3	3
Complete_Patterns	1	1	1
Concatenate	3	3	3

```
*****************************************************************************
* Testing all paths of  Subprogram: Alternate
*****************************************************************************
-------------------------------------------------------
 Test conditions case 10 of 27 for subprogram: Alternate



Test conditions required for test case 10 are:

 210: Set (Right = null or else Right.Variant = Bad ) to False
 212: Set (Left.Variant = Bad  ) to False
 215: Set (Left.Variant = Alt  ) to True
 216: Set (Right.Variant = Alt  ) to False
 238: Set (New_Left.Variant = Rng  ) to True
 239: Set (New_Right.Variant = Rng and New_Left.Name = New_Right.Name  ) to True

-------------------------------------------------------------
Statements to be executed during test case 10 are:

 195:  Procedure Alternate is
 207:  Begin
 210:  If Right = null or else Right.Variant = Bad  then
          *** Condition is False
 212:  Elsif Left.Variant = Bad
          *** Condition is False
 214:  End if    -- for 210
 215:  If Left.Variant = Alt   then
          *** Condition is True
 216:  If Right.Variant = Alt   then
          *** Condition is False
 230:  Else
 231:   New_Left := Right;
 232:   New_Right := Left;
 233:  End if    -- for 216
 237:  End if    -- for 215
 238:  If New_Left.Variant = Rng   then
          *** Condition is True
 239:  If New_Right.Variant = Rng and New_Left.Name = New_Right.Name   then
          *** Condition is True
 240:  Return Merge_Ranges(New_Left,New_Right);
 252:  End
```

Figure F-1. TestGen Conditions for Path Testing ALTERNATE

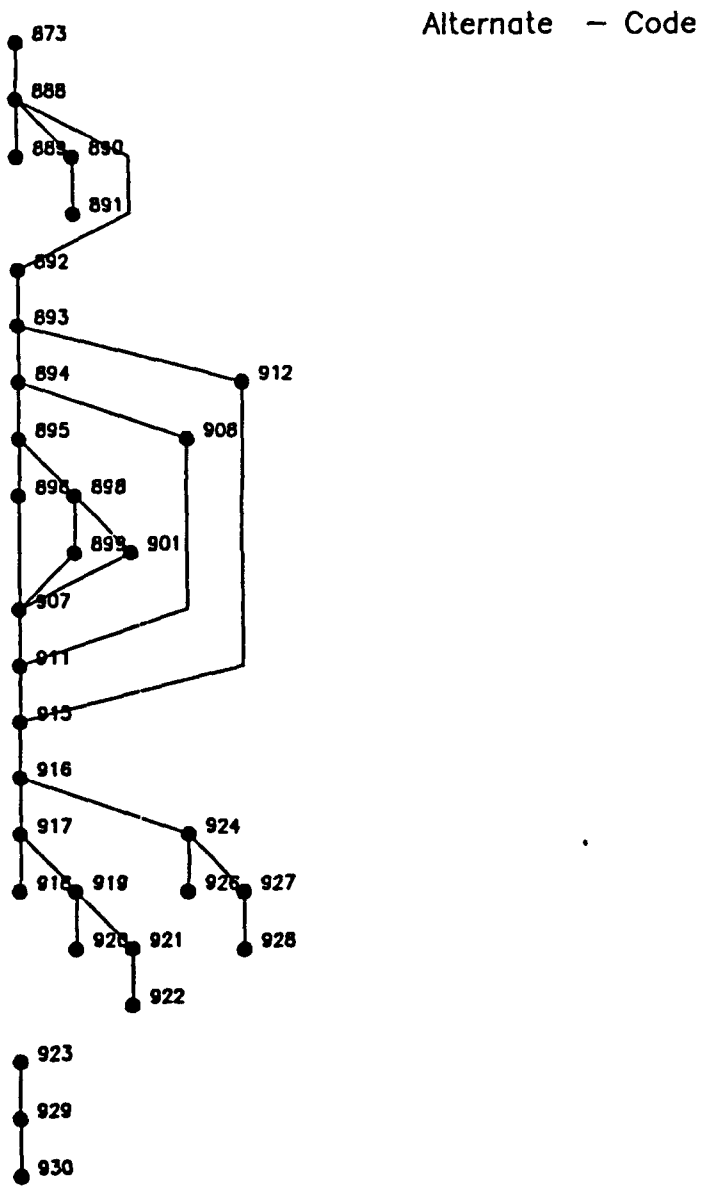


Figure F-2. TestGen Example Control Graph of ALTERNATE

Distribution List for IDA Paper P-2628

NAME AND ADDRESS	NUMBER OF COPIES
Sponsor	
Lt Col James Sweeder Chief, Computer Resources Engineering Division SDIO Room 1E149, The Pentagon Washington, DC 20301-7100	2
Other	
Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dr. Dan Alpert, Director Program in Science, Technology & Society University of Illinois Room 201 912-1/2 West Illinois Street Urbana, Illinois 61801	1
IDA	
General Larry D. Welch, HQ	1
Mr. Philip L. Major, HQ	1
Dr. Robert E. Roberts, HQ	1
Ms. Ruth L. Greenstein, HQ	1
Mr. Bill R. Brykczynski, CSED	65
Ms. Anne Douville, CSED	1
Dr. Richard J. Ivanetich, CSED	1
Mr. Terry Mayfield, CSED	1
Dr. Reginald N. Meeson, CSED	2
Ms. Katydean Price, CSED	2
Dr. Richard L. Wexelblat, CSED	1
Ms. Christine Youngblut, CSED	2
IDA Control & Distribution Vault	3